1ST EDITION

Java Web Internals

Unlock the secrets of Java web servers, frameworks, and application architecture

FRANCISCO ISIDRO MASSETTO

Foreword by Bruno Souza,
Java Champion | International Mentor and Author

Java Web Internals

Unlock the secrets of Java web servers, frameworks, and application architecture

Francisco Isidro Massetto

‹packt›

Java Web Internals

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Kunal Chaudhari
Relationship Lead: Rithika Shetty
Project Manager: Ankit Anilkumar Maroli
Content Engineer: Deepayan Bhattacharjee
Technical Editor: Irfa Ansari
Indexer: Hemangini Bari
Production Designer: Aparna Bhagat
Growth Lead: Vinishka Kalra

First published: April 2026

Production reference: 1140426

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83588-148-4
www.packtpub.com

To my loved wife Luana and my children Nicolas, Luca, and Enrico

Daddy loves you all

Thanks for being by my side during this journey

Foreword

You're debugging a performance issue in production. Your AI assistant suggests three different solutions. Each one looks reasonable.

Now... Which one do you choose?

AI doesn't know your system's architecture constraints. It can't see the memory footprint implications. It has no context about your team's deployment pipeline or the technical debt accumulating in your legacy modules.

You do.

This is the new reality of software development. AI will generate more code than ever. Yet, developers are the ones that carry the responsibility for the critical decisions: system architecture, performance trade-offs, scalability patterns, and technical direction.

Now, you can only make those decisions when you understand how things work under the hood. That's exactly why I'm so impressed by Francisco Isidro Massetto's *Java Web Internals*.

That reminds me of the first time I met Isidro...

I'm driving through São Paulo. Sitting beside me, in the passenger seat, is one of the most influential leaders of the Java ecosystem. Patrick Curran, chair of the Java Community Process (JCP), is visiting Brazil.

Over two hours in traffic, we are on our way to meet a university professor known for his passionate teaching style. Patrick is excited about this visit. As part of a joint effort from the JCP and SouJava (one of the world's largest Java User Groups), we are connecting with educators who share our vision for Java development.

The drive feels endless. We go through neighborhoods I'd never seen before, following GPS directions that lead us further into the city's maze. Patrick is restless – I'm secretly hoping this won't be a disaster.

Our destination: a classroom where this professor was building something special.

Professor Isidro, as he was widely known, is an online sensation. Developers everywhere talk about his energetic approach to teaching fundamental concepts. His reputation for making complex topics accessible grows rapidly. Still, nothing prepared us for meeting Professor Isidro in person.

Isidro wasn't your typical academic. Energy radiated from him. His excitement was infectious. Students hung on every word as he explained the importance of Java.

The normally serious Patrick is laughing and enjoying the visit. He is completely captivated. I'm impressed. The event exceeded every expectation.

That was many years ago...

Patrick has since passed away and is deeply missed in our community. Yet the impact of that encounter continues to this day. Isidro became one of my closest friends. His online influence has grown tremendously. He's now a Java Champion and successful entrepreneur. He has built and deployed real systems with his students. Thousands of developers rely on his insights for their daily work.

This book you are holding now, brings back all that energy I witnessed that day in São Paulo. Refined through years of teaching and real-world application. Starting with web fundamentals, you'll first understand the basic concepts of client server applications and then build your own web server. Then you'll implement a mini application server (think of it as your personal Tomcat). Finally, you'll create your own web framework (let's say, your own mini-Spring).

By the end, you'll understand the architectural decisions and trade-offs behind every major server and framework in the market. And more – you'll become a powerful developer.

When AI suggests a caching strategy, you'll know the memory implications. When it recommends a threading model, you'll understand the concurrency trade-offs. When it proposes a framework pattern, you'll recognize the performance characteristics.

This deep knowledge isn't just *helpful* in an AI-driven world. It's *essential*.

Isidro brings the same passion and energy I witnessed firsthand, distilled into practical knowledge you can apply immediately.

The future belongs to developers who understand the power of AI for sure. Yet, the best developers of the future have a deep appreciation for the fundamentals that make great software possible, and also deeply enjoy the development craft.

This book, built on of Professor Isidro's beliefs of the importance of fundamentals and on having fun developing software, gives you both.

Bruno Souza

Java Champion

Founder of SouJava

Mentor at Code4.Life

Contributors

About the author

Francisco Isidro Massetto, a.k.a. Professor Isidro, is Java Champion with a PhD in Computer Engineering. He works as a professor, lecturer, and entrepreneur, bringing in-depth technical knowledge to computer science students through both in-person classes and his YouTube channel.

With over 25 years of teaching and industry experience, he has always sought to create links between academia and the job market.

He is never content with just *using* the technology itself and always tries to study in depth how technologies are built – to learn more and understand how to use them better.

I would like to express my heartfelt gratitude to my beloved family. My wife and children, in addition to supporting me in writing this book, were extremely understanding when I was working during times when we should have been together.

About the reviewer

Harish Padmanaban is a Senior IEEE Member and Chair of the IEEE Houston Section with over 13 years of experience in Java & Python application development, web servers, distributed systems, and enterprise architecture. His expertise spans servlet containers, application frameworks, request lifecycle management, and scalable backend design.

He has authored multiple technical books and holds four global patents in software architecture and automation-driven systems. Harish actively contributes to the global engineering community as a reviewer for IEEE and peer-reviewed journals, combining strong theoretical foundations with hands-on system design experience.

"Understanding the internals is what transforms a developer into an architect."

— Harish Padmanaban

I would like to sincerely thank the Packt team for entrusting me with the technical review of this book. It was rewarding to engage deeply with the architectural concepts and implementation details, and I appreciate the openness to incorporating review feedback to strengthen technical precision and clarity.

My thanks also go to my family for their continued encouragement and support throughout the review process.

Table of Contents

Preface xv

Free benefits with your book xix

Chapter 1: Networks, Protocols, and Packets 1

Technical requirements 2

Exploring the TCP/IP stack with an example 2

The need for application layer protocols 7

Defining a very simple application protocol 8

Understanding sockets 10

Setting up projects with an Eclipse IDE • 15

Setting up a project manually • 16

Step 1: Generate a JAR file for your protocol • 17

Step 2: Compiling and Running your Server project • 17

Step 3: Compiling and Running your Client project • 17

Summary 19

Chapter 2: Workings of a Web Server – A Case Study 21

Understanding an HTTP request 23

Understanding an HTTP response 27

See in action 31

A quick reflection on performance 32

Summary 33

Chapter 3: Implementing a Simple Web Server 35

Technical requirements 36

Starting our project 36

Handling HTTP requests 40

Handling HTTP responses **42**
Performance issues **50**
Summary **52**

Chapter 4: Web Servers vs Application Servers 55

SimpleWebContainer architecture **57**
Handling business rules via HTTP **59**
Implementation issues **62**
Summary **63**

Chapter 5: Deep Dive into Simple Web Container Implementation 65

Creating WebContainerLib classes **67**
Exploring classes on SimpleWebContainer **70**
A small example **75**
Testing • 80
Further implementation issues **81**
Summary **81**

Chapter 6: Simple Web Container Improvements and Expansions 83

Simple application server review **84**
Improvements and expansions **84**
Support for multithreading, or even virtual threads • 84
Session maintenance • 85
Support for multiple applications or contexts • 87
Monitoring and logging • 88
Tools for administration and deployment • 89
Potential uses of simple application server **89**
Educational purposes in educational environments • 90
Embedded server • 90
Use in test environments • 91
Low-performance devices • 92
Summary **92**

Subscribe to Deep Engineering 94

Chapter 7: Why Create a Web Framework 95

Why use Tomcat instead of SAS 95

Understanding servlet: The Tomcat foundation • 96

Web frameworks: Abstracting infrastructure with Spring and Quarkus • 97

Understanding internally how a request works in the Spring framework 99

Request Interception and Handler Mapping • 99

Request Dispatching and Handler Adapters • 100

Invoking Controller Logic and Generating the Response • 101

Lifecycle and extensibility • 102

Building our own web framework 103

Use of embedded Tomcat • 103

Handling GET and POST Methods • 103

Object Serialization and Deserialization • 104

Dependency injection (DI) based on the Jakarta specification • 105

Summary 105

Chapter 8: Web Frameworks – The Necessary Abstraction for Professional Development 107

So, what is a framework? 108

Web frameworks • 108

What do Spring and Quarkus abstract from the developer? 109

The role of the Spring Framework • 109

And Quarkus? • 110

Why create our own web framework? 110

More than writing code: understanding the invisible • 110

A journey guided by three fundamental questions • 111

But first: why leave SAS aside and use Tomcat • 111

The SimpleWebFramework architecture: a new look at the request-response cycle ... 112

Browser to HTTP request • 112

Embedded Tomcat: Robust infrastructure, under our control • 112

DispatcherServlet: The heart of the framework • 113

The metadata extraction module: Understanding the application from annotations . 113

@SimpleController: Identifying controllers • 114

@SimpleGetRequest and @SimplePostRequest: Mapping the service methods • 115

@SimpleBody: Interpreting the request body • 115

@SimpleService: Organizing reusable components • 116

@SimpleInject: The magic of dependency injection • 116

Summary 117

Chapter 9: Detailing the SimpleWebFramework Implementation 119

The main class: SimpleWebFrameworkApplication 120

The embedded tomcat dependency • 121

Framework initialization • 121

Explaining the code step by step • 122

The framework initialization cycle • 124

Benefits of embedded tomcat • 124

The ClassExplorer class: locating the Application's controllers 124

The role of ClassExplorer in initialization • 125

Class scanning with the Depth-First search (DFS) algorithm • 125

Using reflection to identify annotations • 127

Directory structure and search behavior • 127

Registering the found controllers • 128

The importance of automated discovery • 128

The SimpleDispatcherServlet: dynamic routing and On-Demand instantiation... 129

The relationship between ClassExplorer and DispatcherServlet • 129

ControllersMap and RequestControllerData structure • 130

Internals of the SimpleDispatcherServlet • 131

The role of data structures in routing • 134

On-Demand creation of controller instances • 134

Handling HTTP GET and POST methods • 135

Reading the request body with @SimpleBody • 135

Consolidated flow • 136

Summary 136

Chapter 10: The Emergence of Dependency Injection and Its Application in SimpleWebFramework 139

Brief history: the challenge of dependencies in enterprise applications 139

The birth of enterprise JavaBeans (EJB) • 140

The types of enterprise JavaBeans • 141

Session beans • 141

Message-Driven beans (MDBs) • 141

Entity beans • 141

The role of the EJB container and XML configuration • 142

From XML configuration to annotation: the maturation of IoC • 143

Dependency injection: the power of specification 144

Bean types and structural decoupling • 144

Implementations and compatibility: spring and quarkus • 146

Spring framework • 147

Quarkus • 147

A common base, multiple expressions • 148

Implementing dependency injection in SimpleWebFramework 149

CDI as a conceptual base and inspiration • 149

The @SimpleService annotation: defining managed beans • 150

The role of ClassExplorer in service discovery • 151

The @SimpleInjected annotation: the injection point • 152

The DependencyInjectionMap and the instance lifecycle • 153

Instantiation strategies • 154

Instantiation on demand • 154

Instantiation at startup • 154

Integration with the DispatcherServlet lifecycle • 155

The role of decoupling and modular architecture • 156

Summary 157

Chapter 11: Creating the First Application with SimpleWebFramework **159**

Creating the MyWebTestApplication project **160**

The main class: MyWebTestApplication.java **161**

Controllers and route mapping in SimpleWebFramework **163**

Services and dependency injection in SimpleWebFramework **168**

The service interface: defining the contract • 168

The service implementation: linking the business logic • 169

On-demand injection • 170

Understanding the didactic role of DI in SimpleWebFramework • 172

Field injection vs. constructor injection • 173

Paths for SimpleWebFramework expansion **173**

Chapter 12: Unlock Your Exclusive Benefits **177**

Unlock this Book's Free Benefits in 3 Easy Steps **178**

Other Books You May Enjoy **182**

Subscribe to Deep Engineering **185**

Index **187**

Preface

This book is born with a clear, yet challenging, proposition: to guide you through a complete journey of building a web framework from its most basic fundamentals. In a scenario where modern frameworks abstract almost all the complexity of web development, many developers end up using powerful tools without understanding what really happens behind the layers of abstraction. Here, we will take the opposite approach. Instead of starting at a high level, we will build this knowledge from the ground up, understanding each piece that makes up the web ecosystem.

Throughout my career as a university professor, I have always dedicated myself to teaching the fundamentals of Computer Science to my students. This has sparked in me, and in my students, a fascination with understanding technology beyond the surface.

And this subject was no different. Web development is currently much easier thanks to the frameworks available, especially considering the Java ecosystem.

However, web frameworks are nothing more than abstractions of something that has always existed in web specifications since their earliest versions. Today, when we use a framework like Spring or Quakus, for example, much of what we use is the result of abstraction layers that, in the past, we did almost manually.

Throughout this journey, you will be guided from the most fundamental concepts of network communication, through the use of sockets, to the implementation of a functional web server. This approach not only strengthens technical understanding but also allows you to develop a critical view of how distributed applications actually work. After all, every web application depends on clear rules of communication between client and server, based on well-defined protocols, as discussed throughout the book.

With this foundation established, we will move on to building an application server, significantly expanding our capabilities. Unlike a traditional web server, which only delivers static content, an application server is capable of executing business logic and generating dynamic responses based on received requests. This transition represents one of the most important moments in the book, as it's where we move from simply serving files to actually processing information and building complete applications.

To make this possible, a deep understanding of how the HTTP protocol works will be essential. Throughout the chapters, you will see how requests are structured, how responses are assembled, and how every detail – from headers to the message body – influences the

application's behavior. More than just using HTTP, you will implement it in practice, understanding its nuances and limitations, which provides a much more solid mastery of web development.

All this construction will be done using Java, exploring everything from its most basic concepts to more advanced features. We will start with fundamental language structures and essential APIs, such as sockets and I/O, gradually evolving to the use of reflection, annotations, and architectural patterns that allow the creation of more sophisticated solutions. This progression was carefully planned to ensure that each new concept builds upon the previous one, creating a consistent and natural learning curve.

By the end of the journey, you will not only have built a functional web framework, but you will also have developed a deep understanding of how real frameworks operate internally. This means that when using tools like Spring, Quarkus, or any other modern framework, you will be able to see beyond the abstraction, understanding their design decisions, their strengths, and their limitations.

This book, therefore, is not just about writing code. It's about training a developer with a systemic vision, capable of understanding, building, and evolving solutions with autonomy and confidence. If you want to move beyond the superficial level and truly master the inner workings of web development, this journey is for you – and I hope you enjoy it!

Who this book is for

This book is for computer science students and Java Web developers at all levels (junior to senior) who want to know more about Web fundamentals and explore framework internals. The main purpose of this book is to uncover some aspects of how a framework can abstract many communication issues and HTTP protocol manipulation, giving autonomy to developers to focus on business rules implementation.

What this book covers

Chapter 1, *Networks, Protocols and Packets*, introduces the fundamental concepts of network-based communication with a simple client/server calculator implementation.

Chapter 2, *Workings of a Web Server – A Case Study*, concerns the main aspects of web server internals, including how it handles the HTTP protocol, decodes requests, and generates responses in different media types.

Chapter 3, *Implementing a Simple Web Server*, shows, step by step, how to implement the full web server delivering static content.

Chapter 4, *Web Servers vs Application Servers*, discusses the main differences between static and dynamic content generation, and which aspects to evolve in the web server to create an application server.

Chapter 5, *Deep dive into Simple Web Container implementation*, explores not only the Application Server implementation, but also a minimal library to create an application that uses this Application Server as communication infrastructure.

Chapter 6, *Simple Web Container improvements and expansions*, creates a roadmap for expansions and improvements of our Web Container, including multithreading, logs, and so on.

Chapter 7, *Why Create a Web Framework*, discusses the motivation to develop a framework, which abstractions we can achieve with a framework, and how complex it is.

Chapter 8, *Web Frameworks – The Necessary Abstraction for Professional Development*, explains the whole framework architecture and needed components.

Chapter 9, *Detailing the SimpleWebFramework Implementation*, explores implementation details and project decisions to make a simple but functional web framework.

Chapter 10, *The Emergence of Dependency Injection and its Application in SimpleWebFramework*, discovers how a specification can be applied to our framework.

Chapter 11, *Creating the First Application with SimpleWebFramework*, explains how to create your web application using the developed web framework with all dependencies and make it functional.

To get the most out of this book

- You need knowledge of the Java programming language, object-oriented programming, and some network concepts.
- This book is for educational purposes, but it doesn't mean it's restricted to an academic scope. You can use this knowledge in your professional projects.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Java-Web-Internals`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781835881484`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: " Let us explore the definition of our `Request` class in this code snippet:"

A block of code is set as follows:

```
Class Request.java
public record Request(Double op1, Double op2, String oper) {
}
```

Any command-line input or output is written as follows:

```
javac -d bin *.java
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: " To answer these questions, we will initially consider a "classical model" in the literature: the **client-server communication model**."

> **Note**
>
> Warnings or important notes appear like this.

> **Tip**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

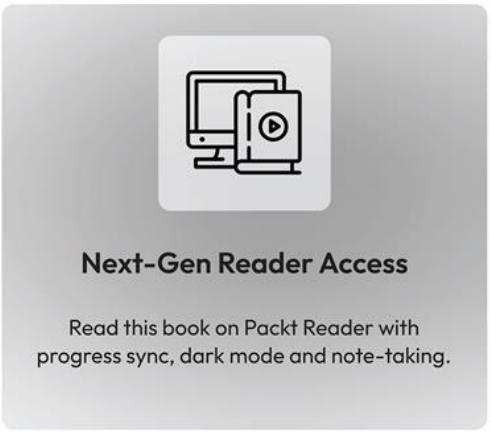
If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:

7-Day Packt Library Access
Get 7-day unlimited access to 8,000+ books and videos. No credit card required.
Available for first-time Packt+ trial users only.



How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Join us on Discord!

Read this book alongside other users, developers, experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the authors via Ask Me Anything sessions, and much more. Scan the QR or visit the link to join the community.

`https://packt.link/deep-engineering-java`

Share your thoughts

Once you've read *Java Web Internals*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

https://packt.link/r/1835881491

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Networks, Protocols, and Packets

The development of web applications has never been as popular as it is currently. Web systems, applications, and IoT devices are gaining more space in people's daily lives.

Therefore, the training of developers to master these technologies, languages, frameworks, and tools becomes essential – developers who are not only capable of delivering solutions to end users, but who also have in-depth knowledge of how this entire communication infrastructure works and enables this development.

For this reason, it is necessary to explore the entire infrastructure, communication models, architecture, and purpose of the services that are used to make this web environment increasingly available to end users.

Understanding how a web server works, the difference between a web server and an application server, the protocols involved, the handling of requests and responses, and the frameworks and their functioning is of paramount importance for your career. Current applications are developed as a set of pieces to be fitted together. For example, when you create a web application, lots of frameworks and libraries do the demanding work for you (database connection, some queries, request handling, response generation, etc.). But what about when things don't work? Is the error in your logic, in the misuse of the framework, or in the poor understanding of a communication protocol?

Finally, if you want to be an above-average developer, you will need to understand how technologies work behind the scenes. And that is exactly the purpose of this book: to show, step by step, how we can implement a "simple" web server, an application server, and a web framework, guaranteeing productivity, reliability, and speed in the development of your projects and applications.

Welcome to this journey, which will explore, demystify, and clarify the entire web infrastructure so you can become a developer well above the average.

In this chapter, we will cover the following main topics:

- TCP/IP stack
- The need for a protocol definition
- Implementing a simple protocol using sockets

By the end of the chapter, you will understand how two applications running on different computers connected by a physical channel (a wired or wireless network) can exchange data. It's more complex than writing "Hello" to another user on a chat application. It requires several layers that control and handle different aspects to make this communication viable.

Note

Your purchase includes a free PDF copy + code bundle

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and additional exclusive extras. See the *Free benefits with your book* section in the *Preface* to unlock them instantly and maximize your learning.

Technical requirements

To understand the purpose of this chapter, we created a repository on GitHub to cover the technical aspects of protocol definition. The code corresponding to this chapter can be found on GitHub here:

`https://github.com/PacktPublishing/Java-Web-Internals/tree/main/Chapter01`

Java 25 is used to implement this code, and you can choose your preferred IDE, such as Eclipse, IntelliJ, or NetBeans.

Exploring the TCP/IP stack with an example

The TCP/IP stack is present in your day-to-day life, and you use connected applications extensively – your messaging app, the websites you visit, streaming apps, and more. The question is: what do they have in common?

The answer is simple, but the use can be complex and very sophisticated. They all use one or more communication channels to transfer data and, therefore, implement communication protocols. But how can this be explored in our reality? Let's understand how these techniques let us develop efficient and robust applications.

It all starts with the interconnection of computer networks (yes, your smartphone is also a computer), and because of this, we are able to develop applications that make use of this

interconnection so that you can send and receive data, regardless of the medium (metallic cables, optical fiber, or even radio waves – the famous wireless).

We need to understand that for two different entities to exchange information, a series of rules need to be established. These rules range from the specifications of the sequences of bits and bytes that are transmitted, their respective meanings, and the actions that are taken depending on what is sent and received, among other things. We are talking about *communication protocols*.

Formally, a communication protocol is a set of rules, procedures, formats, and actions that are specified so that two or more entities can exchange information with each other. Of course, for this to happen, we need to ensure that many steps are completed, and therefore, it is necessary to separate responsibilities and ensure that certain parts of this entire process can be interchangeable and replaced.

What am I talking about? Let's think about a simple example.

Imagine a situation where the president of a Russian company wants to talk to the president of a Pakistani company to develop a new product together. How would this communication take place?

If both spoke in their respective local languages, neither of them would understand anything the other said, and so no product would be made.

So, let's think: what if we adopted a "communication protocol" between them? What if we used two translators who could respectively translate from Russian to English and from Urdu to English? Would it be easier? Certainly, translators could transmit the respective messages and translate them back to their interlocutors. Well, through this type of rule, we manage to establish communication between two different entities, as both follow this procedure and can understand what each one means. Furthermore, all communication between two entities takes place through packets. Imagine a conversation between two people: each phrase in this conversation can be considered a packet. One of these people wants to tell several facts to its interlocutor. If the text is too long, it can be sliced into phrases. Each phrase can be (metaphorically) equivalent to a packet. Formally, a packet is the smallest unit of transmission between two entities. An exceedingly long message (a download of a file) is fragmented into several packets so that we can have control over the transmission of each fragment (checking whether the fragment was transmitted and received in full, order of the fragments, confirmation of receipt of a given fragment, among others).

Now that we conceptually understand what a communication protocol is, let's turn to the more technical part of network communication today. Today, all communication is done through a stack of protocols (protocols responsible for rules at different levels). There is a reference model called **Open Systems Interconnection (OSI)**. This is a model created by the **International Organization for Standardization (ISO)** and defines that communication between two entities

is subdivided into seven different layers, each with its own responsibility and manipulating specific rules at each stage of communication.

The OSI model is the main model in the computing field for defining the communication standard between two distinct entities. Even though it is a reference model, it is fundamental to understanding the separation of responsibilities in each layer.

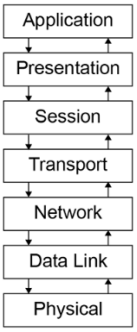


Figure 1.1: The seven layers of the OSI model

Let's understand each of the layers:

- **Layer 1: Physical** – This layer is responsible for the rules for transmitting digital signals in a physical medium (cable, optical fiber, or waves). Here, the main concern is how a set of bits is transmitted through these transmission media and at what transmission speeds.
- **Layer 2: Data Link** – In this layer, the concern is to establish data delivery rules between two entities already connected by a physical medium. Among the functionalities of this layer are the identification of the peripheral that connects your entity (your computer's network card or your smartphone's antenna), ensuring that the datasets were transmitted correctly, through parity calculations, among others.
- **Layer 3: Network** – Here, the protocol deals with issues related to identifying the machine that is connected to this network (it may have several communication peripherals – network cards, for example), its addressing, as well as the interconnection of several networks (the connection between your home's local network and the access provider's network), and which routes will be drawn for the message to leave its origin (your laptop) and reach its destination (the website you want to access, for example).
- **Layer 4: Transport** – In this layer, the concern is whether the data will arrive consistently or not. Here, we can have two types of treatment: connected and unconnected protocols. In connected protocols, the delivery of the message, any retransmissions if the message has not been delivered, and the order of the messages are

of crucial importance (an example of this is a website that needs to deliver all its content so that the browser can display the page on the screen). In non-connected protocols, concerns about possible packet losses (message fragments) are not so crucial. An example of this is streaming or video transmission, where any losses do not compromise the transmission (you just stop seeing or hearing a small part of your media)

- **Layer 5: Session** – At the session layer, the rules are aimed at coordinating two connected applications. . For example, on a conference call, participants can define their conversation rules, such as: if they can talk at the same time or one of them talks and another one just listens, or if no one says anything for a period, the call finishes automatically. The OSI session layer provides a channel between entities that keeps communication active or capable of working if communication is broken without having to start it over.
- **Layer 6: Presentation** – In the presentation layer, the main function is to define how the data will be presented or exchanged. For example, we can exchange data between two entities defining a common character encoding standard. In other words, the presentation layer can be a "translator" between data format representations.
- **Layer 7: Application** – Here, we can find the most popular applications used on the internet, such as HTTP for web browsing, SMTP for email exchange, and FTP for file transfer. The application layer provides functions and interfaces that applications use to communicate over a network.

In this model, it is worth remembering that a given layer uses functions from the layer immediately below when transmitting data, and when receiving data, it delivers them to the layer immediately above. While the perception is that each layer of an entity communicates directly with the corresponding layer of the target entity, communication occurs from the application level to the physical level. Information is transmitted and received at the physical layer of the other entity, then delivered successively until it reaches the application layer, as shown in the following figure.

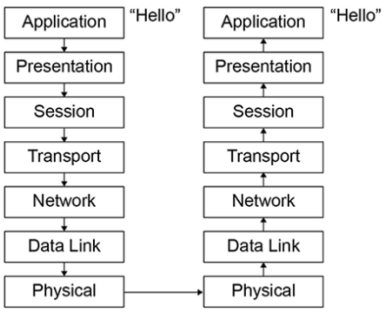


Figure 1.2: Communication between layers

Now, in real life, this seven-layer model is implemented in a much simpler way, with a more compact model with only four layers, in addition to the application layer (where we can add functionality from the session, presentation, and application layers). The alternative model to OSI is the TCP/IP model, which has become the standard protocol of the internet. TCP/IP has the following layers:

- **Media Access layer** – Encompassing the physical and data link layers of the OSI model
- **Network layer**
- **Transport layer**
- **Application layer** – Encompassing the application, presentation, and session layers of the OSI model

Each layer's functions are the same as in the OSI model, and the TCP/IP model is currently the most used in connected applications, as it has two protocols in the transport layer. *Figure 1.3* depicts the equivalence between the TCP/IP and OSI reference models:

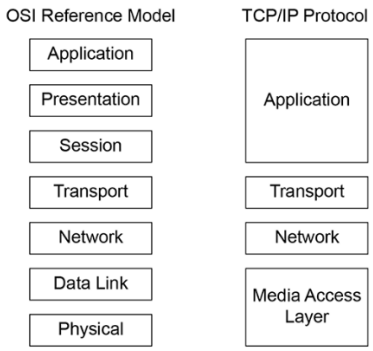


Figure 1.3: Equivalence between the OSI reference model and TCP/IP protocol

Considering the transport layer, TCP/IP has two protocols:

- **Transmission Control Protocol (TCP)** is a reliable transport protocol that guarantees delivery and ordering of transmitted packets. Because it is reliable, each packet carries a lot of control information, adding additional overhead and impacting the effective payload size. It is widely used in most current applications, especially web applications, which we will discuss throughout this book.
- **User Datagram Protocol (UDP)** is an unreliable transport protocol. This is a protocol that tolerates transmission failures. Due to this characteristic, little control information is needed in each packet, which means that the same packet compared to the TCP protocol has an increased payload size. This is very useful in real-time applications, such as video calls or Voice Over IP. Some online games use UDP as a protocol, such as Xbox

Cloud Gaming and NVIDIA GeForce Now, as well as meeting tools (Zoom, Google Meet, or Microsoft Teams), and so on.

As we can see, protocols in both the seven-layer conceptual model and their implementation in the TCP/IP model are fundamental for building reliable applications. From this point on, we will focus on developing examples using the TCP connection-oriented protocol, as it provides the infrastructure for all communication between web servers and browsers.

It is important to note that this protocol stack is available and integrated into our operating systems. If you must implement an application that communicates with another, any operating system will provide a set of libraries your programming language will use. All this complexity is built in and available behind the scenes. You, as a programmer, do not need to access this entire protocol stack.

The need for application layer protocols

Okay, we understand how everything works, but what if we could implement our own application protocol? How should we think about an application? How do we plan a connected application?

To answer these questions, we will initially consider a "classical model" in the literature: the **client-server communication model**. This model will guide all our work throughout this book.

Basically, the application is divided into two entities: the client entity, responsible for requesting the information/resources it needs, and the server entity, responsible for providing these resources to the client. The client is an application that usually (but not necessarily) interacts with users through a proper interface (graphical or not) and needs data that is not available on it. So, it requests data from another entity: the server. For example, when you browse the internet, you are looking for information not available on your computer: news, articles, videos, or code hosted on other computers.

The server will always be a passive entity, which waits for the client to trigger communication and then respond. The client is responsible for activating the server by requesting data. We will call every request from the client a **request** and every return from the server a **response**. This way, our client/server model works with request/response-based communication, as shown in the following figure:

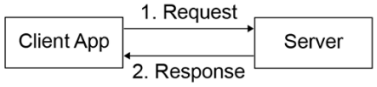


Figure 1.4: Client/server request/response

Bringing this model into everyday life, the browser you use to access a website is your client, and the web server that delivers this content to be displayed in the browser is your server. Basically,

the request is the address to be contacted (later, this will be much more detailed), and the response is the content of the page to be displayed.

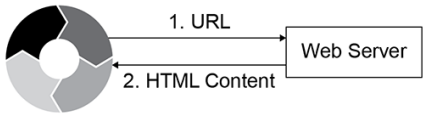


Figure 1.5: Browser/web server

Before exploring more complex aspects of communication between browsers and web servers, let's start a little simpler: let's understand how to fully define an application protocol between a client and a server.

Defining a very simple application protocol

Let's assume we are going to implement a remote calculator, initially, with the following six arithmetic operations:

- Sum
- Subtraction
- Multiplication
- Division
- Power
- Square root

In this case, to simplify things, let's define the structure of our request. Of the six operations, five expect two operands, and only one (square root) expects a single operand. Let's define a standard structure:

- `Request:`
- `operand1: number`
- `operand2: number`
- `operator: text`

An example of a sum operation can be viewed in the following code snippet. In defining the protocol that will be used, we will transmit serialized Java objects. This notation is a textual notation (in this case, the JSON format); however, the communication will be done by sending and receiving serialized Java objects.

```
Operation 2 + 3 → Request:
{
```

```
    "operand1": 2,
    "operand2": 3,
    "operator": "+"
}
```

Now, as an answer, you might be thinking that just returning a number with the result of the operation is enough, right? However, let's consider some restrictions:

- The division cannot have the second operand with the value of `0`
- The square root cannot have the first operand with a negative value
- We will not accept any operator other than those defined (`+`, `-`, `*`, `/`, `^`, and `sqrt`)

Therefore, if we send data that violates some of these restrictions, our protocol needs to signal that the operation cannot be performed. Also, if we send a symbol other than the supported operators, we have to signal that the operation is not recognized.

So, our answer is as follows:

`Response:`

- `status: text`
- `value: number`

We will now see a table where our protocol handles valid and invalid requests.

Operation	***Request***	***Response***	***Detail***
2 + 3	`{ "op1": 2, "op2": 3, "oper": "+" }`	`{ "status":"Ok", "value": 5 }`	Successful operation.
3 / 0	`{ "op1": 3, "op2": 0, "oper": "/" }`	`{ "status":"Invalid", "value": null }`	Operation failed. One of the operators violates the division constraint.

5 ** 8	<pre>{
	"op1": 5,
	"op2": 8,
	"oper": "**"
	}</pre>
		<pre>{
		"status":"Unsupported",
		"value": null
		}</pre>
			Operation failed. Informed that operation is not supported.

Once we understand this concept, we can translate it into a programming language. But first, we need to understand how to make this communication possible through sockets!

Understanding sockets

Sockets are operating system resources for communicating information between remote applications. They abstract the addressing of an application within the operating system and provide a bidirectional channel for this information exchange. To do this, we need to understand that there are two types of sockets:

- **Connected sockets** use TCP as a transport protocol, ensuring communication delivery and reliability
- **Unconnected sockets** use UDP as a transport protocol, providing no guarantee of delivery

Note

The focus of this chapter is not to exhaust the subject of network communication using Java. It is just to show an example of how we can create a way for two entities to communicate through a connected protocol (TCP). If you want to delve deeper into network programming concepts, an excellent reference is the book *Learning Network Programming with Java: Harness the hidden power of Java to build network-enabled applications with lower network traffic and faster processes*, by Richard M. Reese, available in the Packt catalog. Also, if you are interested in Java fundamentals, a great reference is the book *Learn Java 17 Programming: Learn the fundamentals of Java Programming with this updated guide with the latest features, Second Edition*, by Nick Samoylov, also available in the Packt catalog.

In Java, the use of TCP sockets is simplified, allowing you to create very robust applications.

Observe the following figure, where we apply the concept of connected sockets to a client/server application.

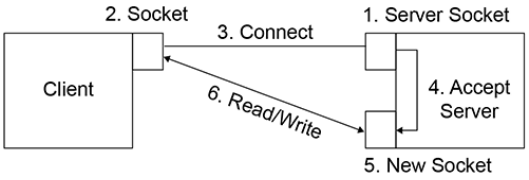


Figure 1.6: Socket and server socket

For us to correctly identify a process on a machine, we need two fundamental pieces of information – the machine's IP address (or its hostname) and the port:

- The **IP address** identifies only the machine we want to connect.
- The **port** identifies which application we want to communicate with. As a modern operating system can run multiple applications simultaneously, the port (which ranges from `1` to `65535`) uniquely identifies the process on the machine)

In our example, `CalcServer` (as we will call our server application) creates a `ServerSocket` (step 1 in the preceding figure) by assigning it the port on which it will wait for connections. This function is exclusive to this class. The client application (which we will call `CalcClient`) also creates a socket (step 2) and assigns the remote server address (IP address and port) for connection (step 3). `CalcServer` receives the connection request and accepts it (step 4), creating another socket, which will be responsible for the effective exchange of messages (step 5), exactly as in the following scheme. Finally, both entities can send and receive messages by reading from and writing to sockets (step 6).

So, let's implement the code, starting with our communication protocol.

In this case, two classes are necessary: the `Request` class and the `Response` class to enable communication operations. Note that both classes implement the `java.io.Serializable` interface so that objects are converted to byte arrays when transmitted over the network. This is because, in this example, we are using a Java application to communicate with another Java application. If we eventually want something more flexible, allowing clients developed in other languages to communicate with our server made in Java, we will choose to serialize the objects to text (for example, a JSON format) rather than to bytes, to increase compatibility.

These conversions (from objects to byte arrays, and vice versa) are called **marshaling** and **unmarshaling**. We have the following details:

- **Marshaling** – When you have an object or a data structure, you can encode it to a byte array (or a character array – a string).
- **Unmarshaling** – You receive a byte (or character) array and parse it, decoding it to another structure; easier to handle in your code.

Let us explore the definition of our `Request` class in this code snippet:

```
Class Request.java
public record Request(Double op1, Double op2, String oper) {
}
```

Now, we can explore our `Response` class definition in the following code:

```
Class Response.java
public record Response(String status, Double value){
}
```

Now, let us understand how `CalcServer` works. In general terms, every server has a common behavior: it works in an infinite loop, accepting requests, manipulating their data, and generating responses, exactly like in the request/response model we saw earlier. The following algorithm depicts its behavior.

```
Create server socket to listen and bind it to a port number
while (true) {
Accepts new connections to a new socket
Reads data from new socket and assign to a request object
Creates a response object
Handle data (in this case analyzing operator and   operands
If operator is supported and operands are correct, performs the operation and sets
the response status to "OK" and the operation result
If an operand is incorrect (e.g., division by zero), sets an "Invalid Operand
error" on response and result is null
If an operand is not supported, sets an "Unsupported Operation" status and result
is null
Sends the Response object to sender
}
```

The complete source code of this server is available in the following GitHub repository: `https://github.com/PacktPublishing/Java-Web-Internals/tree/main/Chapter01`.

If we analyze the source code, we can quickly identify each algorithm step. Every step is commented in the source code, and you can identify them easily:

```
// step 1
ServerSocket serverSocket = new ServerSocket(8350);
// step 2
```

```
while (true) {
    // step 2.1
    Socket socket = serverSocket.accept();
    // step 2.2
    ObjectInputStream in = new ObjectInputStream(
    socket.getInputStream());
    Request req = (Request)in.readObject();
    // step 2.3
    Response rep;;
    // step 2.4 - handling data
    switch(req.getOper()) {
        // other cases to handle ...
        case "/":
            // step 2.5
            if (req.getOp2() != 0) {
                rep = new Response("Ok",req.op1()/req.op2());
                rep.setStatus("Ok");
                rep.setValue(req.getOp1()/req.getOp2());}
            // step 2.6
            else {

                rep =  new Response("Invalid", null);
            }
            break;
        default:
            // step 2.7
            rep = new Response("Unsupported", null);
    }
    // step 2.8
    ObjectOutputStream out = new ObjectOutputStream(
    socket.getOutputStream());
    out.writeObject(rep);
}
```

Finally, let's explore our `CalcClient` class code. Its objective is only to interact with the user, receiving (according to the desired operation) the value of each operand, assembling the request, sending it, and waiting for the response to finally display it. The following algorithm describes the Client application:

1. Reads data from the input (operation and first operand).
2. Depending on the math operator, reads the second operand.

3. Creates a `Request` object with that data.
4. Connects to the `Server` application.
5. Sends a `Request` object.
6. Receives a response.
7. Prints the result.

The complete source code is available on GitHub at `https://github.com/PacktPublishing/Java-Web-Internals/tree/main/Chapter01/CalcClient/`, and we can identify each step as follows:

```
void main(){
    // step 1
    String oper;
    Double op1, op2=null;
    oper = scanner.nextLine();
    op1 = Double.parseDouble(scanner.nextLine());
    // step 1.1
    switch(oper) {
        case "+":
        case "-":
        case "*":
        case "/":
        case "^":
            op2 = Double.parseDouble(scanner.nextLine());
            break;
    }
    // step 2
    Request req = new Request(op1, op2, oper);
    // step 3 - If you know your IP Address, you can use
    //          it. It proves the packet actually hits
    //          the whole network stack
    Socket socket = new Socket("localhost", 8350);
    // step 4
    ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
    out.writeObject(req);
    // step 5
    ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
    Response rep = (Response)in.readObject();
    // step 6
```

```
        System.out.println(rep);
    }
```

Okay, the code is understandable. But how do we implement this in practice?

In Java, if you use an IDE such as Eclipse, NetBeans, or IntelliJ, I suggest you create three different projects:

- One to maintain the classes that define your protocol (this could also be a library)
- One for `CalcServer`
- One for `CalcClient`

The key point is to reference, in both projects (`CalcServer` and `CalcClient`), the project (or library) that contains your protocol classes. This way, you maintain a single code base (without duplication) and can make communication between the server and the client compatible.

Setting up projects with an Eclipse IDE

You must import each project separately in your Eclipse IDE (protocol, server, and client). To make the server and client use the protocol classes (`Request` and `Response`), you configure the build path:

Project Properties → Build Path → Configure Build Path

On the screen, go to the **Projects** tab, and in the `Classpath` item, add the `CalcProtocol` project, as depicted in the following figure.

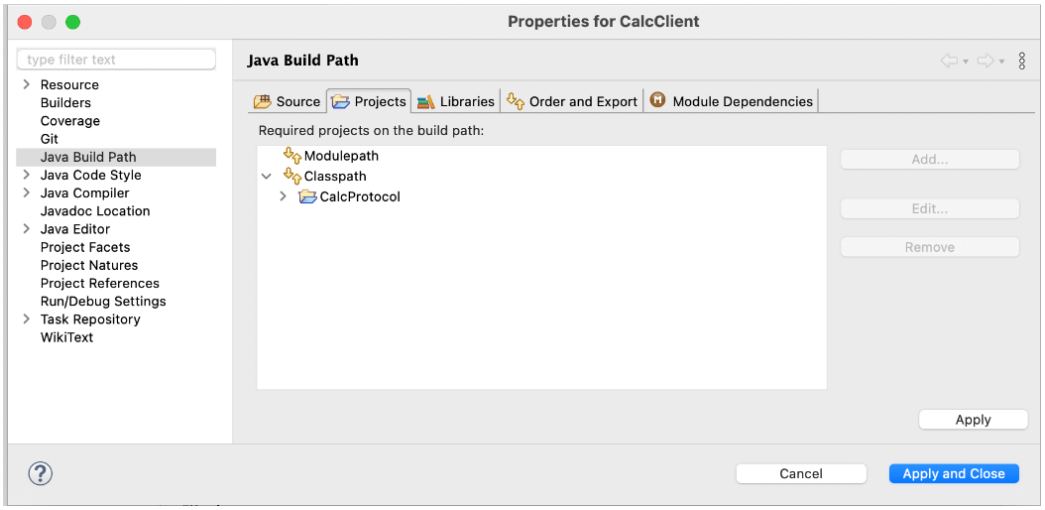


Figure 1.7: Configuring the project build path on Eclipse

Remember that you must do the same operation on the `CalcServer` project.

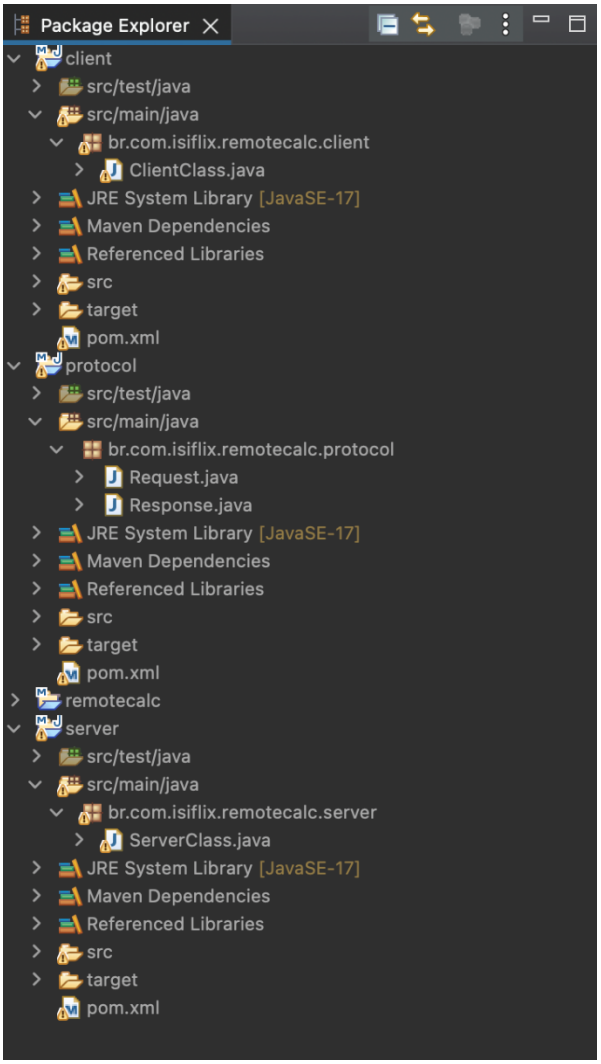


Figure 1.8: Project structure on Eclipse

Setting up a project manually

If you are not using an IDE to develop your applications, you can do it manually using command-line tools. It is necessary for you to have practice with CLI tools (bash, Windows command line, PowerShell, and others).

The protocol project (which contains Request and Response classes) is a common module for the other two projects (Server and Client). If you want to reuse it, the best alternative is to generate a JAR file for this project as follows. Step 1: you must go to your project.

Step 1: Generate a JAR file for your protocol

1. Go to your protocol project root folder.
2. Create (if it does not exist) a folder for your compiled classes (let's call it "bin").
3. Compile your Java files (assuming that your files are in the **src** folder).

```
javac -d bin *.java
```

4. Generate a JAR file containing your compiled classes

```
jar –create –file protocol.jar -C bin
```

Step 2: Compiling and Running your Server project

1. Go to your protocol project root folder and create your **bin** folder.
2. Compile your Java classes (this step is not necessary since Java 11, but if you want to ensure that your code is correct before running, you can do this). In this step (regardless of which way you choose), you must reference your protocol JAR file in the Classpath parameter.

```
javac -cp <your protocol jar file> -d bin src/*.java
```

3. Once your files are compiled, you have to run your Server main class (here we will assume that the filename is ServerClass).

```
java -cp <your protocol jar file> bin/ServerClass
```

Step 3: Compiling and Running your Client project

Previously, open another terminal window.

1. Go to your protocol project root folder and create your **bin** folder.
2. Compile your Java classes (this step is not necessary since Java 11, but if you want to ensure that your code is correct before running, you can do this). In this step (regardless

of which way you choose), you must reference your protocol JAR file in the Classpath parameter.

```
javac -cp <your protocol jar file> -d bin src/*.java
```

3. Once your files are compiled, you have to run your Server main class (here we will assume that the filename is ClientClass).

```
java –cp <your protocol jar file> bin/ClientClass
```

Now, regardless of which option you choose, let us run our applications. You must first run the `server` project (because it creates the socket that listens for connections), followed by the `client` project to interact with the user and send/receive messages.

The following figures show both application behaviors. *Figure 1.9* depicts the server showing no console interaction, waiting for the client's request:

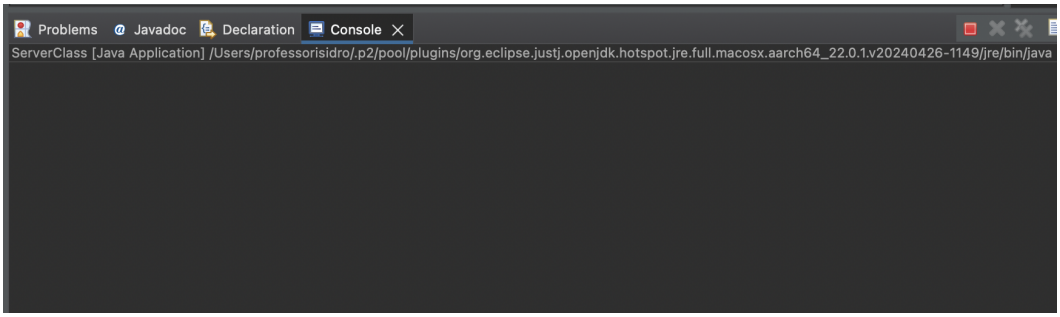


Figure 1.9: Server running showing no messages waiting for the client's request

Figure 1.10 shows the client interacting with the user, who inputs the operation (+) and two operands (3 and 8).

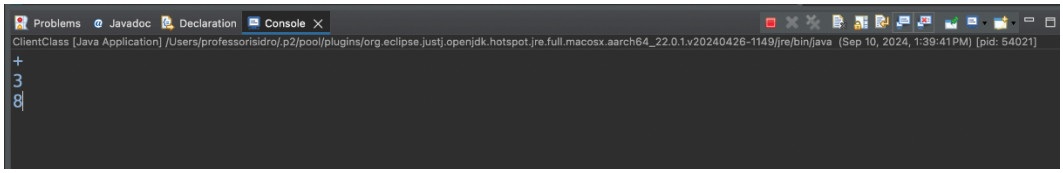


Figure 1.10: Client interacting with the user

Figure 1.11 depicts the server receiving the client's request.

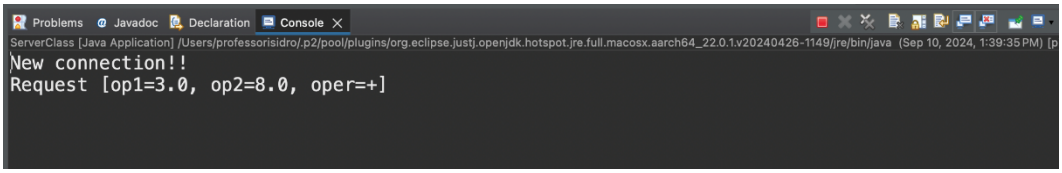


Figure 1.11: Server receiving client's request and printing messages on the console

Figure 1.12 depicts the client printing the result output.

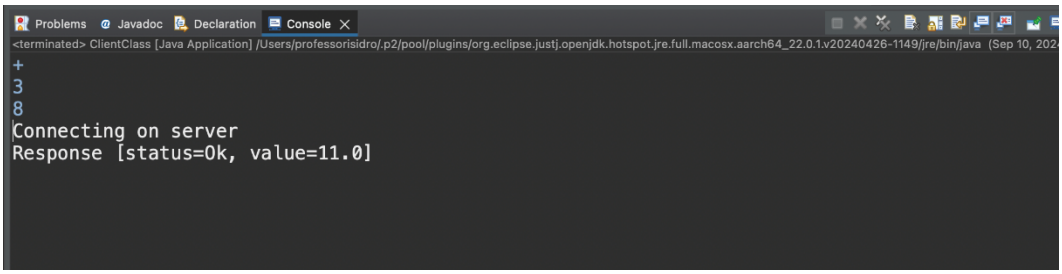


Figure 1.12: Client receiving server's response and outputting the result

Summary

In this chapter, we learned about the fundamentals of communication between processes on different machines. The existence of a communication protocol is essential. In this case, we were able to understand the OSI reference model and the most popular implementation, TCP/IP. Furthermore, we created a client-server application where we were able to define everything from the protocol rules to its Java code.

In the next chapter, we will explore how the HTTP protocol works and start modeling our own web server to understand how web communication works.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

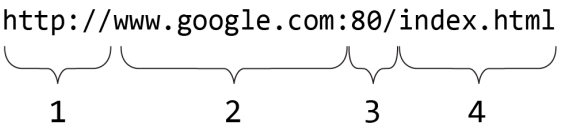
2

Workings of a Web Server – A Case Study

In our daily lives, we interact with web servers extensively, often without realizing it. Whenever we access websites and portals to read news, search for information, or look for images and other media, we rely on web servers.

Whenever you are using a browser (Google Chrome, MS Edge, Opera, or others) and browse the internet, you are certainly accessing resources hosted on a web server.

We can summarize it very simply by saying that a web server is basically a static file server whose resources are accessible through an element called a **Uniform Resource Locator (URL)**. That is, it indicates, within a hierarchical structure (folders), which resource (file) will be accessed on the web server. Every URL has four components:



Some of them are mandatory, and others are optional for the user. Let's explain each one:

1. **Protocol**: This indicates the "language" in which the browser and web server will exchange information. In this case, we have the HTTP protocol, which we will discuss in detail later.
2. **Server address**: Here, we can either use an IP address with its four groups of values (from 0 to 255) or a domain name (the famous **Domain Name Service** – **DNS**). This type of service maintains databases of server names and their respective IP addresses, allowing the user to not have to know the IP address of a service.

3. **Port**: The port, in this case, as we have already seen, identifies the service on a machine (since we can have several servers on the same machine). However, this is an optional component, since according to the browser's default configuration, if you omit this parameter, port 80 is automatically defined. If you ask yourself why a web server's port is 80, the answer is that we have a set of services with standardized numbers registered by a global organization called the **Internet Assigned Numbers Authority** (**IANA**) (`https://www.iana.org/`).

4. **Resource**: Here, we indicate in a hierarchical way (in folders) the file we are looking to access. Again, this is an optional component because if not specified, the web server sends the default resource previously configured on it. Here, there is a variation, which can inform us that access to the resource can be parameterized.
   In what way, you ask? Imagine that you want to recover a blog page that omits the comments made in the publication. Note that it is not a different page. It's just the same page but without a certain section (the comments section). To inform this type of parameterization, in the URL itself, after the name of the resource, the character (`?`) is included, and then a list of parameters with their respective values (parameter=value) is added. This list can contain as many parameters as necessary and is separated by the delimiter (`&`).

   An example can be seen here:

   `http://mywebsite.com:80/post.html?comments=off`

   Another example here is a case of multiple parameters: `http://mywebsite.com:80/post.html?comments=off&author=true`

In a certain way, we can summarize that a web server is a file server that delivers its respective contents through a request in the format of a URL, as in the following diagram:

Figure 2.1: The whole process of a browser requesting a file from a web server

Every response from a web server can either be in text format or in bytes, such as for images, audio, or video. Therefore, it's important to delve deeper into understanding how this entire protocol operates.

To do so, we will cover the following main topics in this chapter:

- Understanding an HTTP request
- Understanding an HTTP response
- A quick reflection on performance

Let's begin!

Understanding an HTTP request

In general terms, **HTTP** stands for **HyperText Transfer Protocol**.

When we type a URL in the browser, a request is created in text format and sent to the web server.

Let's understand in detail what this request looks like.

Suppose we had a web server running on our local machine (`localhost`) on port `8080` (just to specify a different port). Our browser would have the following URL:

`http://localhost:8080/`

The complete corresponding request looks like this:

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="124", "Google Chrome";v="124", "Not-A.Brand";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "macOS"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: pt-BR,pt;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: _ga=GA1.1.1261657737.1624022434;
_ga_97XYS46373=GS1.1.1697464106.5.0.1697464106.0.0.0
```

In this request, we can identify four major components:

1. **The first component**: The main request, that is, `GET / HTTP/1.1`
   Here, we have the browser requesting the root resource (/) using the standard `GET` method, indicating that this request will be made using the HTTP 1.1 protocol. If you wanted, for example, another resource such as `page.html` inside the `/public` folder, then we would have the following:

   ```
   GET /public/page.html HTTP/1.1
   ```

   But is there only the `GET` method for a request? The answer is *no*. There are several HTTP methods, and they are basically divided between HTTP methods that are suitable for sending information in the body of the request and methods that are not as suitable.

   In these two categories, we can highlight the following:

   - `GET`: The method that only considers the search for a resource (as if we were querying content).
   - `POST`: When we send additional information to the web server on the request (for example: data filled on a form page). We will explore the `POST` method a bit more after explaining the header.

2. **The second component**: The host destination of the request; that is, `Host: localhost:8080`
   Here, we have the server address (either IP address or name) and the corresponding port, indicating whom this message is intended for.

3. **The third component**: Request headers. Headers are request metadata that influences internal implementation.
   For example: we can inform on request headers' authentication data (to identify a user) the source address, the software that performed the request, and many other things:

   ```
   Connection: keep-alive
   Cache-Control: max-age=0
   Upgrade-Insecure-Requests: 1
   User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/
   537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36
   Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
   avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
   Sec-Fetch-Site: none
   Sec-Fetch-Mode: navigate
   ```

```
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: pt-BR,pt;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: _ga=GA1.1.1261657737.1624022434;
_ga_97XYS46373=GS1.1.1697464106.5.0.1697464106.0.0.0
```

Here, we can see parameters that identify who the requester is (`User-Agent`), the operating system, the type of accepted response (`Accept`), and the type of connection (`Connection`), among many others. These parameters complement the request, but for now, we don't need to delve into the details.

4. **The fourth component**: A blank line.
   The blank line has a purpose because, as the number of headers can be variable, this line indicates the end of the request. Therefore, the web server reads the HTTP request line by line until it finds this indicator.

   But what about methods that send information, such as the `POST` method, for example? Let's look at an example. Suppose we want to send a request and, with this request, a simple text, such as a letter. What would the request be like?

```
POST /form HTTP/1.1
Host: localhost:8080
User-Agent: ...
Accept: ...
Content-Type: text/plain
Content-Length: 294
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis augue lectus,
elementum nec fermentum quis, aliquam a sapien. Vivamus et urna vel turpis
rutrum accumsan consequat vitae ex. Donec tincidunt, arcu vitae iaculis
tempus, sapien eros bibendum est, vitae pretium nunc felis sed libero.
```

   In this request, we can highlight the method to be used (`POST`) requesting the `/form` resource. In addition, we have the inclusion of two other headers:

   - `Content-Type`
   - `Content-Length`

The `Content-Type` header indicates the type of content to be sent. There are several types of content that we can send, and each one is treated differently. Among these types of content, we can highlight the following:

- `text/plain`: This is the most basic content to be sent – plain text without formatting.

`multipart/form-data`: In this case, form data is sent separated by its respective fields and contents. Each field with its content comes with a line that delimits its beginning (it starts with a sequence of dashes followed by a generated hexadecimal number and, finally, the name of the field with its content). This helps to send different types of data (text, image bytes, file bytes) in the same message, as they are all separated by these delimiters. Additionally, the `Content-Length` is crucial to determine the number of bytes to be sent. The web server reads the number of bytes based on this header.

Let's see an example of a form that sends two fields, `login` and `password`, with the respective content: `test@mail.com` and `1234`. This data is described in the `Content-Disposition` section between the delimiters:

```
POST /form HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.64.1
Accept: */*
Content-Length: 247
Content-Type: multipart/form-data;
boundary=------------------------19a411ac4da7de66

--------------------------19a411ac4da7de66
Content-Disposition: form-data; name="login"

test@mail.com
--------------------------19a411ac4da7de66
Content-Disposition: form-data; name="password"

1234
--------------------------19a411ac4da7de66–
```

5. `application/json`: This is a text format but with the purpose of representing an object (Java or another language). The JSON standard is a way of representing objects in a programming language with their attributes and values, delimited by the { and }

characters. A small example of a Java object with the `"id"`, `"description"`, and `"price"` attributes with their respective attributes can be seen here:

```
{
    "id":1,
    "description": "Computer",
    "price":1000.0
}
```

Obviously, there are other HTTP methods, such as `PUT` and PATCH (also used to send data with the intention of updating records), and DELETE, whose intention is to delete resources (this is widely used in the development of web APIs using REST, which we will see in Part 3 of this book, Mastering Your Skills with Jakarta and Java Evolution). This chapter's main objective is to explain the operation of a simple request that expects some content as a response and a more complex request that requires data to be sent more robustly.

Now that we understand the structure of a request, let's analyze the structure of an HTTP response in more detail.

Understanding an HTTP response

Now that we have a request, how can we understand how an HTTP response behaves?

Here's an example HTTP response to a valid request:

```
HTTP/1.1 200 OK
Location: http://localhost:8000
Date: Tue, 21 May 2024 13:27:27 GMT
Last-modified: Tue, 21 May 2024 13:27:17 GMT
Content-type: text/html
Content-length: 149

<html>
   <head>
      <title>It Works! </title>
   </head>
   <body>
     <h1>Hello World! </h1>
     <p>Welcome to my Website</p>
   </body>
</html>
```

An HTTP response is also a structure based on three components. Let's get to it:

- **First component – HTTP headers**: We will mainly detail the first line of this header, which is the HTTP status code:

  ```
  HTTP/1.1 200 OK
  ```

  Here, we have an example of a successful response, as the success code (`200`) and its textual form (`"OK"`) indicate that the resource request was found and sent to the client.

  There are a series of HTTP status codes for the most diverse situations in requests. These codes follow a three-digit pattern whose first digit always defines the type of status. Let's analyze this:

  - **1xx**: Statuses related to information about the progress of the request. In other words, every status that starts with the digit 1 indicates that the web server received the request and it was understood (it was correct).
  - **2xx**: Status related to the success of the request. Codes starting with 2 indicate that the request was received, decoded correctly, and accepted for processing.
  - **3xx**: Status related to the user's need for extra actions to complete the request. The most common situation is redirection. For example, the browser tries to connect to a server whose request is redirected to another server.
  - **4xx**: Client-related error status. Here, we have many situations where the client either sent a poorly formed request or is looking for a non-existent resource (a page that does not exist), or even does not have permissions to access that content.
  - **5xx**: Error status related to the server; that is, the server receives the request and understands it, but an error occurs when processing it (the server may also be overloaded with requests, for example).

And here, we can look at some examples that are quite common in our daily lives. Observe the following table:

Status code	**Description**
`200 - OK`	The request was successful because the resource was found and sent to the client.

`404 - Not Found`	The requested resource is not on the server.
`403 - Forbidden`	The client that requested the resource does not have permission to access it (some type of authentication is required for this).
`405 - Method Not Allowed`	When a resource can only be accessed via the `POST` method, for example, and the client sends a `GET` request.
`500 - Internal server error`	The web server, when processing the request, generated an error because it was unable to complete the processing, but it sends a message notifying you of this.

These are just a few examples of HTTP status codes. A complete list can be found at this link: `https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`. This includes the complete specification of the HTTP protocol and its status codes.

Following the HTTP status, we have a variation of headers depending on the web server implementation. However, some of these header components are important for us to analyze (again):

- `Content-type`: Indicating the return type (and this is essential for the browser to know whether it will render HTML content, display plain text, or even display an image on the page). This is important because the browser can display different formats on different ways. For example, lack of content-type for an image can make the browser download it instead of displaying it.
- `Content-length`: Indicating the size (in bytes) of the response. This information helps the browser to understand how many bytes of response it needs to read to obtain the complete content.
  The `Content-type` and `Content-length` header components are present in both requests and responses. These header parameters are fundamental for letting the client decide what to do with the content bytes.
- **Second component – a blank line**, precisely to effectively separate the textual content of the header from the response content.
- **Third component – the content of the response itself**: This content will always be an array of bytes. These bytes can be the characters of the text that is stored in a file (HTML, TXT, CSS, JS, or other) or the bytes that form an image (JPG, PNG, GIF, and so on), the bytes of a PDF file, from a ZIP file, and so on. But how are these different types of media

identified by the browser? Through the header components (the `Content-type` header that will inform the browser what it should do with this received content). In the answer example, the content is an HTML page:

```
<html>
   <head>
      <title>It Works!</title>
   </head>
   <body>
     <h1>Hello World!</h1>
     <p>Welcome to my Website</p>
   </body>
</html>
```

- **Fourth component – another blank line**: Again, we have a blank line to delimit the end of the answer.

Once you understand the structures of requests and responses, you'll see that an HTML page is not just made up of text. Every page has, in addition to the text itself, images, various other media, and references to other files (scripts, style sheets), among others.

How, then, can we handle the entire content delivery of a complete page? Look at the following diagram:

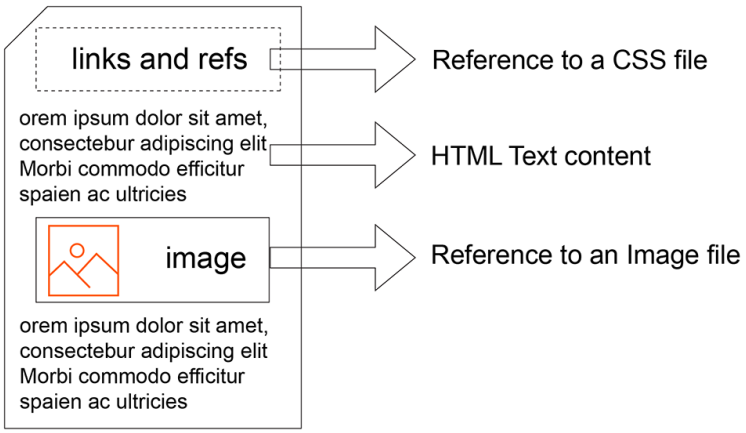


Figure 2.2: Structure of an HTML file with image and reference to CSS file

Let's assume this complete page is a file called `INDEX.HTML`. Inside it, in addition to its text content, there are two references to other files: a CSS file (which we will call style.css) and

another reference to an image (which we will call `image.png`). How is this page rendered in the browser with all these components?

The answer seems simple, and it is: for each different component of the page, we make a new HTTP request.

The justification for this answer lies in the very structure and purpose of the web server's existence. Because a complete page is made up of several files, and the web server is effectively a file content server (whose communication is done via the HTTP protocol), it is logical for the browser, when starting the process of rendering this page, to request each of the additional files until it completes receiving all content.

Thus, we have the following sequence of requests from the browser to the web server:

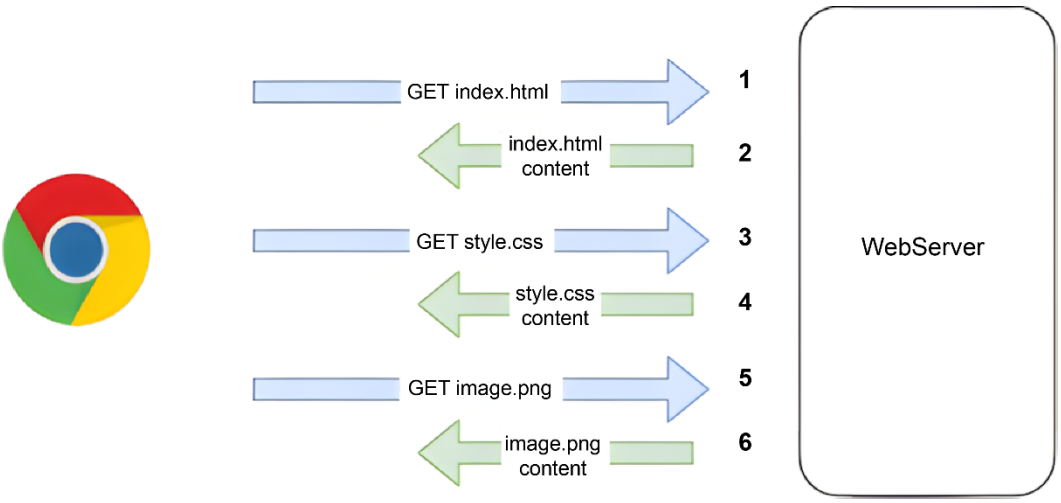


Figure 2.3: Full process of requesting a complete HTML file with embedded images and other files

If we analyze it in a very pragmatic way, we will see that the idea of how the web server works is to be as simple as possible, once we have minimal mastery of manipulating the HTTP protocol. We need to analyze not only the content of requests and responses, but also the control codes, the internal structure of the web server, and the hierarchy of folders that store content to be shared.

See in action

A good way to see all these headers and files being requested by the browser is to go to any website and watch everything happening through Google Chrome "Developer Tools."

First, open your Google Chrome application and go to the menu in the upper right (three vertical dots). Select the "More Tools" menu option, then "Developer Tools." You can also press the F12 key on your keyboard if you are on a Windows machine.

With the Developer Tools section open, go to the "Network" tab. This tab shows the complete behavior of a request and the respective response.

Type your URL and watch all requested resources (files, scripts, images, and so on). Figure 2.4 depicts this behavior in a very simplified way with a local web server. The "index.html" file that is requested leads to three other files (script.js, styles.css, and favicon.ico).

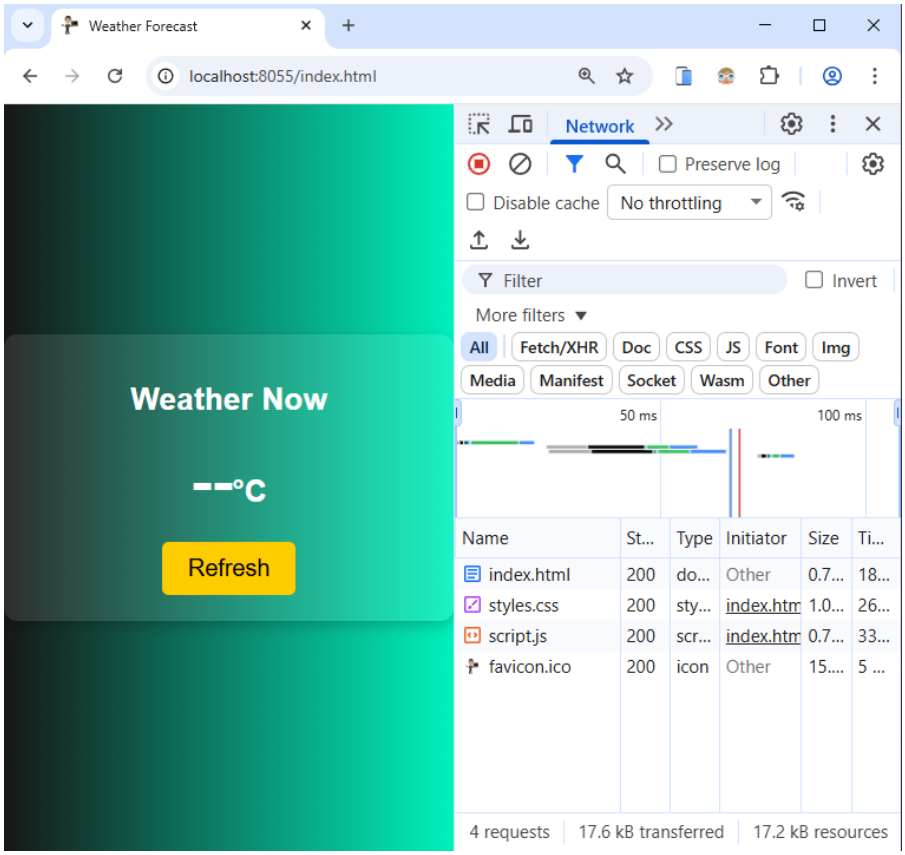


Figure 2.4: A real request for a local webserver leading to several files

A quick reflection on performance

As we saw previously, a web server always works in the request-response model. However, developing a web server can lead to important design decisions. Imagine that every web server keeps receiving requests and sending responses, in an infinite but sequential repetition. As the number of requests increases, we will have a very large queue, causing the web server to become overloaded and resulting in delays in serving customers.

How, then, can we create a web server that is more efficient and can handle multiple requests simultaneously?

In this case, the concept of multiprogramming (multithreading) becomes fundamental for us to achieve a high degree of parallelism. Each request can be handled by a different thread, freeing up the communication channel so that other clients can contact the web server.

Summary

Understanding the structure of an HTTP request and response is essential for its implementation, whether it is a single-threaded or multithreaded web server. In this case, processes for handling the request and assembling the response are necessary to achieve successful development, not only creating the algorithms but also knowing how to identify scalable points to increase the degree of parallelism when serving requests.

In this chapter, we explored the structure of an HTTP request, the internal behavior of a web server, and the structure of a URL. We can now decode both HTTP requests and responses, understanding all involved components (most of them are text-based).

In the next chapter, we will explore a simple web server implementation using sockets and discuss the theoretical background. The main purpose of implementing a web server in Java is to understand how a web server works internally and to verify that the same URL can generate several requests to obtain the content of an HTML page, for example.

3

Implementing a Simple Web Server

As seen in previous chapters, we can understand that to create applications that communicate over the network, we need socket libraries. Sockets are bidirectional communication channels to send and receive data through the network. Furthermore, we also know that sockets transmit bytes, and, because of this, there is a fundamental need to implement our own communication rules (communication protocol), involving not only the order of messages but also their respective formats.

Taking this into consideration, we analyzed a well-known protocol, the HTTP protocol, and discovered that, in addition to its fame, it is relatively "simple" to implement. This is because the HTTP protocol is basically made up of a set of text messages, making them easier to understand and decode.

Therefore, the objective of this chapter is to implement a small version of a simplified web server and put all these concepts into practice. Again, we do not want to create a competitor to Apache Web Server or Microsoft's **Internet Information Services (IIS)**, but to create a small version for teaching purposes and to understand the technical aspects involved.

In this chapter, we'll cover the following:

- Starting our project
- Handling HTTP requests
- Handling HTTP responses
- Performance issues

Technical requirements

Before we start coding our web server, let us list some fundamental characteristics that must be met by our software:

- Support for `GET` and `POST` methods only
- Support for submitting HTML content (pages)
- Support for receiving URL parameters (query params) and request body
- Support for sending content in plain text (TXT files)
- Support for sending images in the following formats:
    - JPG/JPEG
    - PNG
- Handling of successful requests (200 OK response code)
- Handling requests where the resource is not found (404 NOT FOUND), as well as generating a standard response page
- For any other types of handling errors, generate a 500 INTERNAL SERVER ERROR response code and a standardized page showing the error

In this way, we can generate an understanding of the functioning of requests and HTTP protocol responses.

Starting our project

As we already know, our `SimpleWebServer` application (that is what we will call it from now on) is a file server (that is, it provides file contents) requested via a URL. (An entire explanation of this has already been provided in *Chapter 2*.)

Based on this and according to the following diagram, we need to have a folder that will be the root folder, in which all the files that will be accessible by `SimpleWebServer` will be stored. We will refer to this folder as `DocumentRoot`.

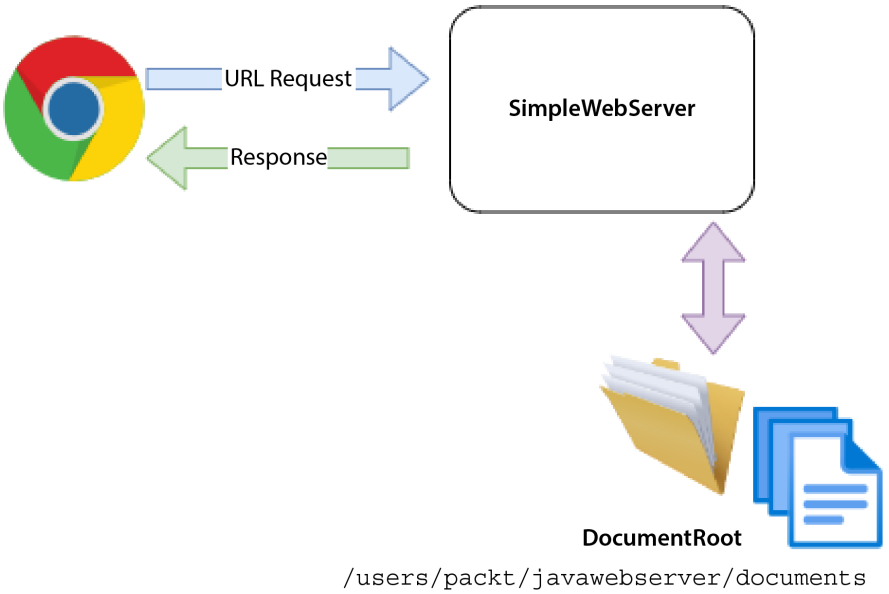


Figure 3.1: SimpleWebServer architecture

This way, we can start our Java project (use your favorite IDE or editor for this) by creating a first configuration file.

```
public class WebConfig {
    public static final String  DOCUMENT_ROOT="<your specific folder>/content";
}
```

Of course, we will talk about refactoring this code. For example, as a simple refactoring, we can suggest reading a configuration file to set the `DOCUMENT_ROOT` variable, instead of setting this variable hardcoded. More complex refactoring may involve the use of threads, for example.

But how would a web server work in terms of code? Let's explore an algorithm so that we have an overview, and then we will detail each of the parts:

```
webserver algorithm
1    define a port number
2    begin
3        create a server socket bound to port number
4        while (true)
5            accepts connection on client socket
6            request handling
```

```
7            send response
8        end
9    end
```

It might seem straightforward, but there's a lot to detail. However, it's crucial to ensure this web server remains simple to avoid excessive processing time and delays in response. This is our primary challenge in implementing a solution such as this.

Now, it's time to start detailing our `SimpleWebServer` application. Oddly enough, the main class at the start contains only a single relevant line:

```
public class MainClass {
void main() {
        WebLogger.welcome();
        WebConfig.setup();
        WebServer webServer = new WebServer(8055);
        webserver.work();
    }
}
```

Do you find the `void main()` part weird? Don't be surprised; this is an "unnamed methods" feature available from Java version 21 (as a preview feature) that allows you to simplify the `main` method header, making it much more enjoyable for beginners—unlike the old `public static void main(String args[])` feature.

But let's get to the point: the first two lines of code display, respectively, a welcome message in the console (you can check this in the source code available in this book's repository) and configure some variables necessary for our web server to work.

The effective line that interests us is precisely the last one: the creation of a `WebServer` object, associating it with port 8055 (in this case, it is an arbitrary port; you can define the port number you want, knowing, of course, that the default port of any web server is port 80).

But as nothing can be that simple, the large volume of code is precisely in the `WebServer` class that we will detail from now on. However, at this point, we will build our class gradually, starting in a very simple and abstract way. As we want to meet the proposed requirements, we will increase the class with new features.

We'll start by implementing the base algorithm.

```
public class WebServer {
```

```
    private int port = 80;
    private ServerSocket serverSocket;

    public WebServer(int port) {
        this.port = port;
        try {
            this.serverSocket = new ServerSocket(this.port);
                    } catch (IOException ex) {
            WebLogger.error("Couldn't initialize
                                SimpleWebServer on port " + port);
        }
    }
}
```

Here, we need to minimally explain the code. In this class, a `ServerSocket` object associated with the specified `port` is created (assigning it to the declared port attribute) and then the work method is invoked. The `work` method `works` as follows:

```
private void work() {
    while (true) {
        try {
            Socket socket = this.serverSocket.accept();
            this.handleRequest(socket);
            socket.close();
        } catch (IOException ex) {
            WebLogger.error("I couldn't handle request from client");
        }
    }
}
```

The `work` method is an infinite loop where the following takes place:

1. It accepts new requests on a client socket.
2. It handles these requests, decoding the messages in the HTTP protocol format.
3. It closes the client socket so that no resources are left open.

And, of course, the most significant part of this method is the call to the `handleRequest` method. This is where we have a large volume of information to be processed. To do this, we will detail how receiving and decoding the request work.

Handling HTTP requests

Initially, to handle a request efficiently and maintain good organizational practices, we will create a new data type: the `Request` class. This class aims to store all information received from the request in a more structured way to facilitate its manipulation and provide direct access to each component. This is much more efficient than dealing with a sequence of lines of text in the request.

```
public class Request {
    private String httpMethod;
    private String resourcePath;
    private String messageBody;
    private String fileExtension;
    private String destinationHost;
    private Map<String, String> urlParameters;
    private Map<String, String> headers;

    // getters and setters
}
```

In this class, some features were also added, such as `getParameter(String parameterName)` and `getHeader(String key)`, to facilitate future manipulation. These methods return, respectively, the value of a given URL parameter and the value of a received header.

Once this class is complete, we can move on to reading the request itself. This reading and decoding is done by the `handleRequest` method. As it is an algorithm whose operation is more mechanical, we chose to describe its main algorithm instead of listing the code. The full content of this code can be found in this book's repository. An overview of the algorithm is provided below:

```
method handleRequest(Socket){
    1. creates a BufferedReader from Socket Inputstream and a Request Object to
keep request data
    2. loop reading line by line all the request
2.1 Extract HTTP Method, HTTP path and URL Parameters
2.2 Extract Destination Host
2.3 Extract all headers and store them on headers map
2.3 If HTTP Method is POST, looks for Content-Length header
2.4 Loop reading Request Body
// all of steps stores data on Request Object
```

```
    3. reading ends when a blank line is found
    4. handles Response to browser
}
```

We can detail in code some parts of section 2 of the algorithm, such as section 2.1:

```
if (tmpLine.startsWith("GET ") ||
    tmpLine.startsWith("POST ")) {
    request.setHttpMethod(extractHttpMethod(tmpLine));
    request.setResourcePath(extractPath(tmpLine));
    extractURLParameters(request, tmpLine);
}
```

This code snippet requires three other methods to extract each piece of information from the first line of the HTTP request. They are `extractHttpMethod`, `extractPath`, and `extractURLParameters`. Let's start with the `extractHttpMethod` method, which very simply just extracts the first component of the main request:

```
private String extractHttpMethod(String requestLine) {
    return requestLine.split(" ")[0];
}
```

Next, the `extractPath` method seeks to extract only the name of the file and the folder in which this file is located, disregarding the URL parameters for another, more specific method.

```
private String extractPath(String requestLine) {
    String path = requestLine.split(" ")[1];
    if (path.contains("?")) { // in case of having parameters
        return path.substring(0, path.indexOf("?"));
    }
    return path;
}
```

Finally, we have the `extractURLParameters` method, which assembles a `HashMap` instance with each parameter and its respective value. In this case, we treat all values as strings, as the respective conversions must be carried out by the respective applications that may manipulate these parameters in the future.

```
private void extractURLParameters(Request request,
                                  String requestLine) {
```

```
        String path = requestLine.split(" ")[1];
        if (path.contains("?")) {
            String parameters = path.split(Pattern.quote("?"))[1];
            String parameterList[] = parameters.split("&");
            for (String pair : parameterList) {
                String singlePair[] = pair.split("=");
                request.getUrlParameters().put(singlePair[0],
                                                singlePair[1]);
            }
        }
    }
```

In a similar way, we continue reading the entire HTTP request line by line, storing each of the headers in the `Request` object, as well as the body of the request if the selected HTTP method is `POST`.

In this case, depending on the type of content sent in the body of the message, we need to handle it differently. If we receive a structure like JSON or even plain text, we just need to read the respective lines.

Finally, we can handle the request in a form-data format; we need a slightly more sophisticated "parser" to be able to extract the information. A little work will be required to correctly decode the values received in the form for future manipulation.

For a simple web server, however, data sent via the `POST` method may be of little use in a first analysis, as, for it to be manipulated and processed, our `SimpleWebServer` application must evolve into an application server (which we will discuss later in this book). An application server has a capability that a common web server does not have: executing code in a programming language. This is essential for us to process data coming from the request via the `POST` method.

Handling HTTP responses

Once our request message is understood and encoded, we need to generate a response to the client (browser).

Before dealing directly with the code, let's first analyze the requirements defined for `SimpleWebServer`: it needs to support different types of files (including text, HTML pages, and images). This support needs to be handled in a unique way, respecting the file types and, mainly, indicating the type of media in the response (it would be unfeasible to have a different algorithm for each type of media). Therefore, we need to have a method that can uniquely deliver the

content of the file by indicating its type in the response header—more specifically, the `Content-Type` header.

We were able to solve this problem in a very simple and efficient way. We just need a table that maps the extension of the requested file to its respective header value. We can detail this here:

File type	**File extension**	**Content type**
HTML files	`.html` `.htm`	`text/html`
Text files	`.txt`	`text/plain`
JPG images	`.jpg` `.jpeg`	image/jpeg `image/jpeg`
PNG images	`.png`	`image/png`

But how do we implement the content of this table? We can use a `HashMap` instance as shown in the following code:

```
Map<String, String> contents = new HashMap<String, String>();
contents.put("txt", "text/plain");
contents.put("html","text/html");
contents.put("htm","text/html");
contents.put("jpg","image/jpeg");
contents.put("jpeg","image/jpeg");
contents.put("png","image/png");
```

Using this strategy, as we want to add new file types supported by our `SimpleWebServer` application, we simply add new entries to this table, such as `application/pdf`, `application/json`, `audio/mp3`, `video/mpeg`, and so on. More details about file types supported on a web server can be found at this link: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types`.

Now, we can detail our method of handling the response. Let's create a `handleOutput` method. This method corresponds to *step 4* of our algorithm (seen in the previous section) and receives two parameters: the socket to which we are going to send the response, and the `Request` object (so that we know which request will be fulfilled).

Let's analyze the code:

```
private void handleOutput(Socket socket, Request request) {
    String completePath = WebConfig.DOCUMENT_ROOT +
                                    request.getResourcePath();
    try (OutputStream out = socket.getOutputStream()){
        if (Files.exists(getFilePath(completePath))) {
            byte[] content =
                      Files.readAllBytes(Paths.get(completePath));
            out.write("HTTP/1.1 200 OK\r\n".getBytes());
            out.write(("Date: " +  ZonedDateTime.now(ZoneOffset.UTC) +
DateTimeFormatter.RFC_1123_DATE_TIME)
                         + "\r\n").getBytes());
            out.write(("Content-Type: " +
WebConfig.contents.get(
         request.getFileExtension()) +
                "\r\n").getBytes());
            out.write(("Content-Length: " +
                           content.length + "\r\n").getBytes());
            out.write("\r\n".getBytes());
            out.write(content);
            out.write("\r\n\r\n".getBytes());
            out.flush();
} else {
            generate404(socket);
        }
    } catch (Exception ex) {
        generate500(socket, ex);
    }
}
```

We start by highlighting that all code is being managed by exception handling. If any exception is thrown during the request processing, a `500 - Internal Server Error` return message will be sent to the client, represented by the `generate500()` method. This method body will be detailed further in this section.

Even though it's a relatively short piece of code, there are several things we can explore. Initially, we specify that we will use the folder defined in the initial configuration as the root folder to start searching for the file.

We then check whether this file exists. If it does not exist, the method `generate404()` is called, which we will detail next, indicating the "404 Not Found" error.

When the file is found, we need to be very careful to correctly construct the HTTP response. Regardless of the type of file (be it text or binary, such as an image), all its bytes are read and stored in an array (through the `readAllBytes` method from the `Files` class and the `java.nio` API).

Even though it is a message composed of several lines, the delimiter between one line and another must be the combination of the carriage return and the new line (`\r\n` combination). This sequence is important, as otherwise, media in binary format will not be displayed correctly by the browser.

It is also necessary to note that the response is sent in the form of bytes. In all answer-writing instructions, at the end, we always use the `getBytes()` method.

In the header corresponding to `Content-Type`, we consult the table that was previously defined, based on the file extension, by using the `contents.get(request.getFileExtension())` call. This method will return the respective stored `Content-Type` value.

With this, we can very efficiently send any type of media to the browser for rendering.

Now, let's quickly explore the `generate404` and `generate500` methods. Both are responsible for error messages; however, they work differently. Let's look at them:

```
private void generate404(Socket socket) {
    try (OutputStream out = socket.getOutputStream()){
        String message = "<html>
                                <body>
                                   <h1>Not Found!</h1>
    </body>
                              </html>";
        out.write("HTTP/1.1 404 Not Found\r\n".getBytes());
        out.write(("Date: " +
                        LocalDate.now().toString()).getBytes());
        out.write("Content-Type: text/html\r\n".getBytes());
        out.write(("Content-Length: " +
     message.length()+"\r\n").getBytes());
        out.write("".getBytes());
```

```
out.write(message.getBytes());
        out.write("\r\n\r\n".getBytes());
        out.flush();
    } catch (Exception ex) {
        WebLogger.error(ex.getMessage());
    }
}
```

The `generate404` method works very closely with the `handleOutput` method because, in the background, a message is generated along with the HTTP header, indicating that the resource was not found. In a similar way, but by passing the caught exception as an additional parameter (to fetch the entire history of exceptions to be displayed on the page), the `generate500` method can be seen next:

```
private void generate500(Socket socket, Exception exc) {
    try (OutputStream out = socket.getOutputStream()){
        StringBuilder str = new StringBuilder();
        str.append("<html>&lt;body>&lt;h1>Internal Server Error</h1>");
        do {
            str.append(exc.getMessage()+ " \n");
            exc = (Exception) exc.getCause();
        } while (exc != null);
        str.append("</body>&lt;/html>");
        out.write("HTTP/1.1 500 Internal Server Error\r\n"
                    .getBytes());
        out.write(("Date: " + LocalDate.now().toString().getBytes())
                    .getBytes());
        out.write("Content-Type: text/html".getBytes());
        out.write(("Content-Length: " + str.toString().length())
                    .getBytes());
        out.write("\r\n\r\n".getBytes());
        out.write(str.toString().getBytes());
        out.write("\r\n\r\n".getBytes());
        out.flush();
    } catch (Exception ex) {
        WebLogger.error(ex.getMessage());
    }
}
```

Now, it's time to test some successful requests and observe the types of files we can display in the browser.

We will act exactly like a normal user accessing our `SimpleWebServer` application.

In this case, in the defined folder, we have four files:

- Three HTML pages (`index.html`, `page.html`, and `example.html`)
- One text file (`text.txt`)
- One image (`Duke.jpg`)

We will carry out some tests gradually.

Test #1 – Access a simple HTML page, without references to images (URL: `http://localhost:8055/index.html`). This gives the following output:

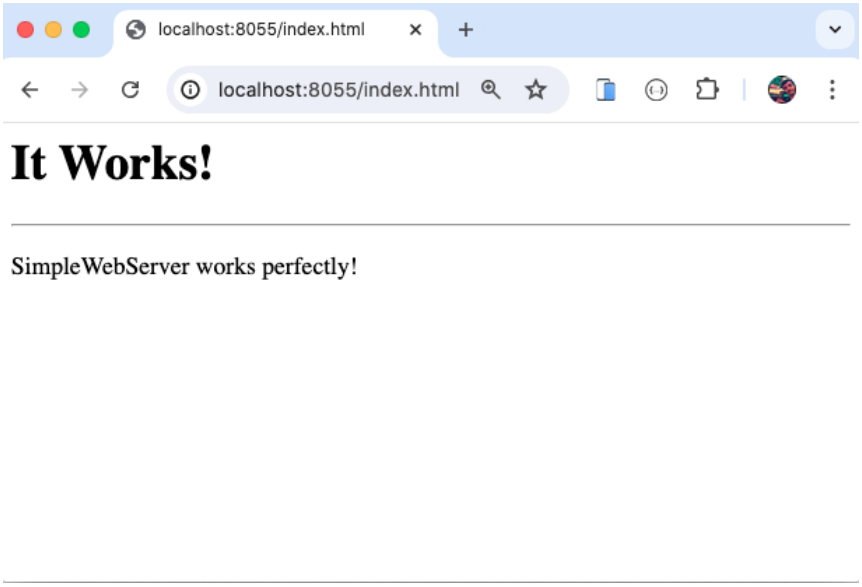


Figure 3.2: SimpleWebServer responding with a simple page to a browser

Test #2 – Access a text file, without HTML formatting (URL: `http://localhost:8055/text.txt`). This gives the following output:

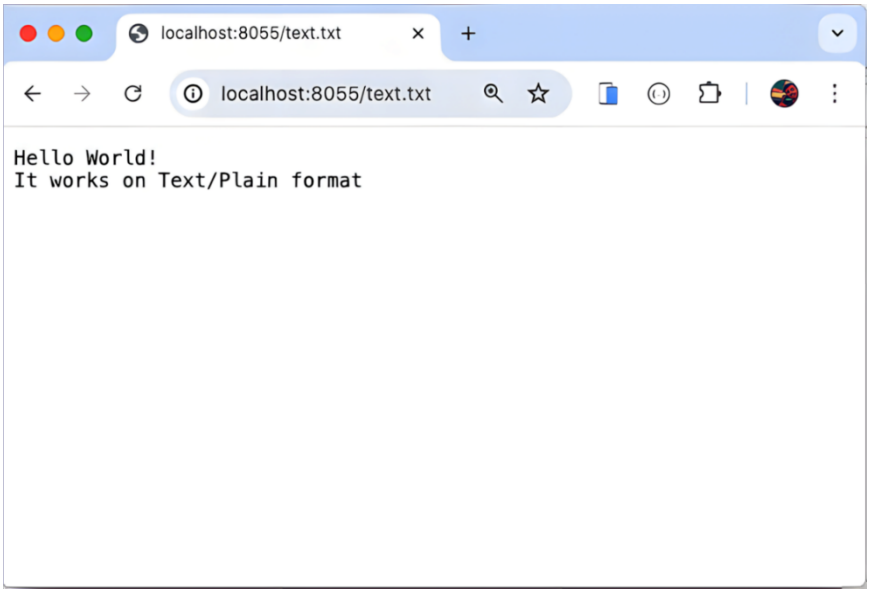


Figure 3.3: SimpleWebServer responding with an unformatted text file (text/plain format)

Test #3 – Access only the image (URL: `http://localhost:8055/Duke.jpg`). This gives the following output:

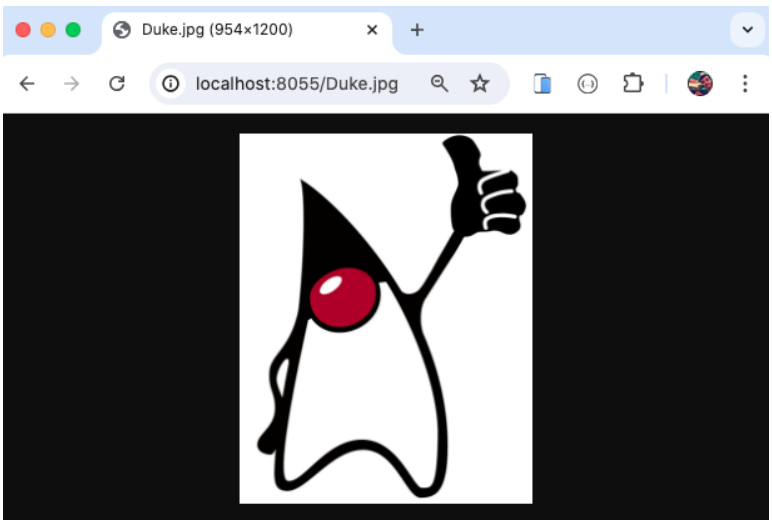


Figure 3.4: SimpleWebServer responding with an image content

Test #4 – Access a page that references an image on the server itself (URL: `http://localhost:8055/page.html`). This gives the following output:

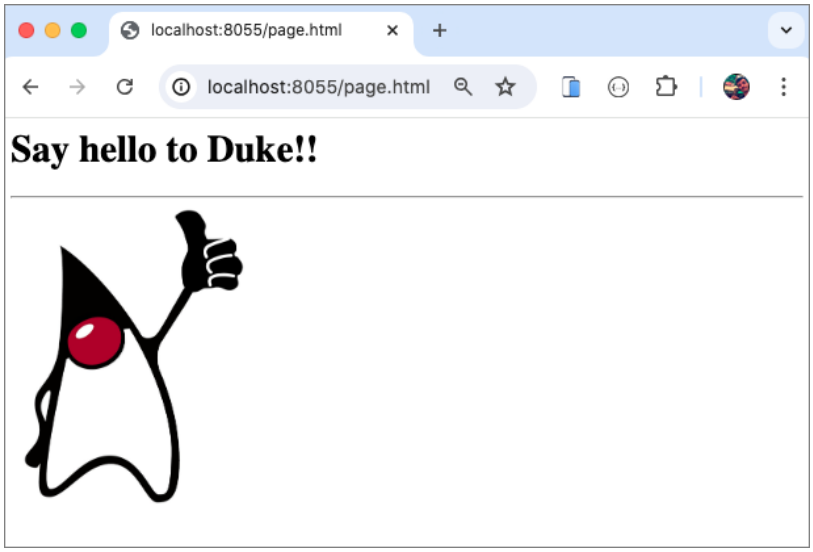


Figure 3.5: SimpleWebServer responding with an HTML page and an embedded image

Test #5 – Access a page that references an external library (Bootstrap). (URL: `http://localhost:8055/example.html`). This gives the following output:

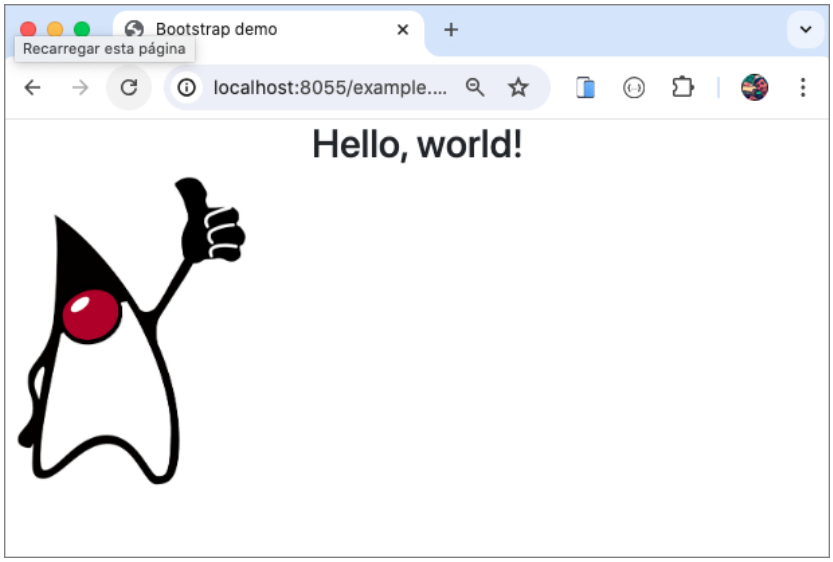


Figure 3.6: SimpleWebServer responding with a page referring to an external library (Bootstrap)

Test #6 – Search for an invalid page (URL: `http://localhost:8055/other.html`). This gives the following output:

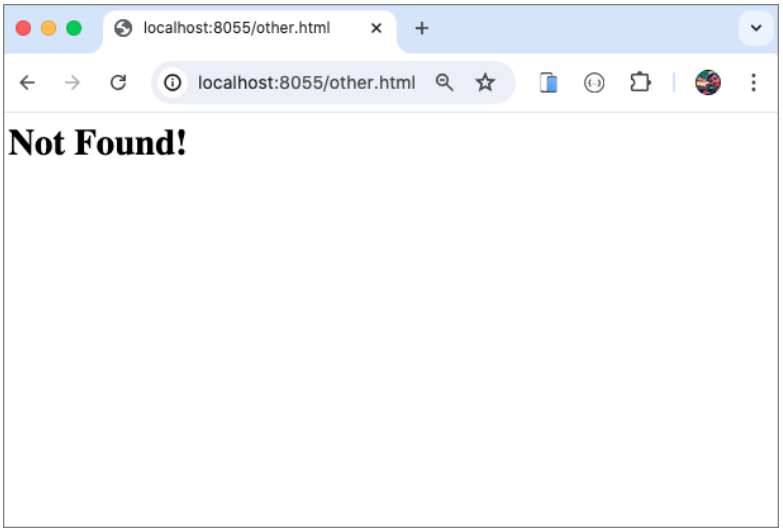


Figure 3.7: SimpleWebServer responding with a "Not Found" message

Thus, according to the premise that a web server is a file server, it must provide the content of each type of document (text, HTML, image, and so on) consistently. Additionally, it should handle pages that include not only text but also images and other resources, such as style sheets, scripts, and more. This ensures a seamless and comprehensive delivery of web content to the end user.

Performance issues

Despite being simple and efficient, our `SimpleWebServer` application works in a single-threaded model; that is, it always serves one request at a time, failing to increase its performance. If there are 10 simultaneous requests, for example, our `SimpleWebServer` application will respond to them one by one sequentially.

This, in a real production environment, makes the adoption of a solution such as this unfeasible. But how can we improve the performance of a web server? One of the main solutions for this is to parallelize the processing of requests.

Java's Threads API can give us a remarkably interesting set of parallelism features. Let us imagine that, when accepting a connection, `SimpleWebServer` delegates its processing to a thread, as in the following diagram:

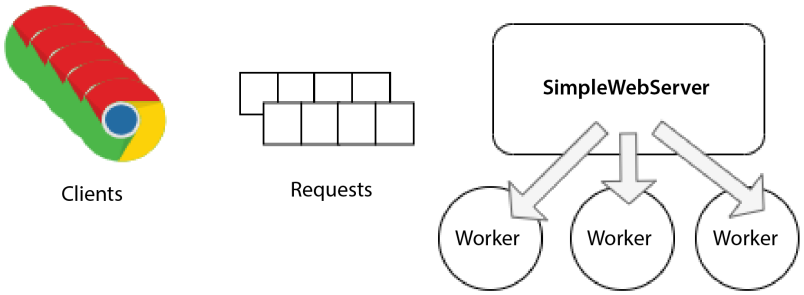


Figure 3.8: SimpleWebServer threads

Each new thread receives the communication socket, handles the request, and generates a response. In our context, it needs to invoke the `handleRequest` method. The functioning of the Java Threads API is based on writing the logic in the inherited run method (from the Thread class), which in our case is just invoking handleRequest. Finally, it is up to the WebServer class, for each request, to create a new thread and execute its start() method, which will create a new thread in the context of the Java Virtual Machine (JVM) and internally call the `run()` method.

We then declare a `WorkerThread` class to perform this manipulation:

```
public class WorkerThread extends Thread{
    private Socket socket;
    public WorkerThread(Socket socket) {
        super();
        this.socket = socket;
    }

    public void run() {
        this.handleRequest(socket);
    }

    // method handleRequest now is a WorkerThread member
}
```

Now, we just need to modify the `work()` method of the `WebServer` class, which, after decoding the HTTP request, instead of invoking the `handleRequest` method, creates an object of type `WorkerThread` and invokes its `start()` method:

```
private void work() {
    while (true) {
        try {
            Socket socket = this.serverSocket.accept();
            new WorkerThread(socket).start();
       } catch (IOException ex) {
          WebLogger.error("I couldn't handle request");
       }
    }
}
```

From the customer's point of view, everything remains exactly as before, even for a page with several elements inside it (images, documents, and so on). Now, the only certainty is that a different thread can execute each of the different requests. With this parallelism, we have two striking characteristics: the first is that we can now serve many requests more quickly and simultaneously (the limitation is due to access to the same filesystem, which can be a bottleneck), and the second is that this `SimpleWebServer` application is implemented following the same strategies as the most used products on the market.

Summary

In this chapter, we learned how to combine the concepts of sockets and all the study of the HTTP protocol we did in previous chapters. As a result, we implemented a small web server that can handle HTTP requests and responses, delivering content from several types of files (text and images, among others). Furthermore, we understood its initial limitation of only being able to handle a single request at a time. Because of this, we implemented a strategy using multithreaded processing to increase the capacity for service requests.

In the next chapter, we will understand how to naturally evolve a web server into an application server. Now, from an HTTP request, we will not only deliver file content, but also deliver the result of the execution of a method as a response. This is where all the dynamic behavior of web applications begins.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

4

Web Servers vs Application Servers

Understanding how a web server works and implementing a version of it allows us not only to master network programming, but also to understand protocols, especially HTTP.

However, when we consider a web server alone, we have a serious problem to deal with: the content provided by the web server is always static. So, looking at a modern-day example, how would we search for information, for example? How would we make a purchase on the Internet, since the browser is one of our main tools today for doing almost everything from wherever we are?

Is it possible, then, to create a web server that, instead of delivering static content such as pages or images, executes the famous "business logic" and responds with the results of these rules?

The answer is yes! It is perfectly possible to manipulate HTTP requests to execute our own rules and send these as responses.

We are talking about application servers.

In general terms, unlike a web server, an **application server** provides the possibility of installing and running applications that also use the HTTP protocol to communicate with their clients.

This means that we can (taking into account that our focus is the Java language) run our own applications and, with this, from an HTTP request, decode it, receive any data (which may come in the body of the request), parameters, etc., execute a method (which may execute a specific service, access databases, perform queries and updates), and, given the response from this method, send it as an HTTP response to the browser. If you are now thinking that e-commerce websites, travel agencies, university web systems, among others, work this way, you are correct.

Application servers allow us to develop applications in the language we master and are responsible for all the parts of receiving and decoding requests and sending HTTP responses. This is similar to what we see in the following figure.

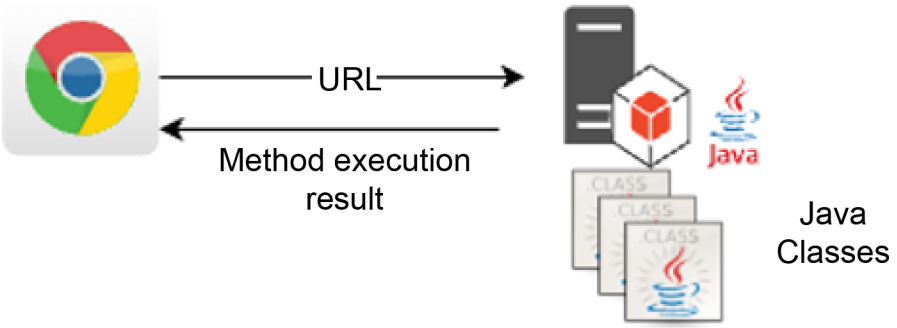


Figure 4.1: How an application server handles URLs

Another characteristic of the application server is that it can also function as a web server in parallel, i.e., an application server can deliver static content (pages and media that we have previously stored) as well as deliver results from the execution of Java classes. This allows for versatility, separating responsibilities: on one side we have a component that manipulates files, and on the other side we have an engine that can dynamically execute Java class methods.

In the entire Java ecosystem, we can have several application servers of varying complexities (IBM's WebSphere, Oracle's WebLogic). However, there is an extremely popular server that does not implement all the specifications of an application server, but for our context, it is an excellent case study for implementation. We are talking about **Tomcat**. Tomcat is not considered a full application server, because to be an application server, it should implement the following items in its specification:

- Enterprise Java Beans (Java objects managed by the Java container that runs the application)
- Data Persistence (such as JPA)
- Transaction Control
- Messaging
- WebServices and REST Development

But if it does not implement all these specifications, why is it so interesting as a case study? Precisely because it is a type of "lightweight application server" whose main objective is to offer dynamic content through an infrastructure of specific classes: servlets. In this way, Tomcat is considered a servlet container, that is, a web server that has features to execute Java classes, in which HTTP requests are handled by these classes.

Our main objective in this chapter is to understand how we can transform HTTP requests into executions of Java classes, so that we can deliver dynamic content through the execution of specific and predefined methods in our Java classes.

What will we explore in this chapter?

- We will define an architecture for our Simple Application Server, which will be called SimpleWebContainer
- We will create a class infrastructure so that we can develop our own applications external to the Application Server, as if we had a set of libraries (simple libraries that can help us avoid logic errors).
- Everything will be based on an already known infrastructure: Java Servlets. We will use very similar concepts so that those familiar with Servlets will understand how our infrastructure works.

SimpleWebContainer architecture

In order to create a version of the SimpleWebContainer by expanding the Simple WebServer, we need to understand that we need to have a segregation of responsibilities. Our Simple Application Server now needs to manage both static and dynamic content.

Because of this, it is necessary to consider some prerequisites.

- The SimpleWebContainer will continue to host and deliver static content of the supported types (HTML, CSS, JS, PNG, JPG and TXT)
- For reasons of simplification of the scope to be developed, aspects of different contexts will not be considered, that is, to isolate different applications. Everything that is executed in the Simple Application Server will be considered from the same (and unique) application context
- We will continue to handle only the GET and POST methods, also to simplify the code.

In this way, we can subdivide our SimpleWebContainer into some components with specific functions and responsibilities, as illustrated below:

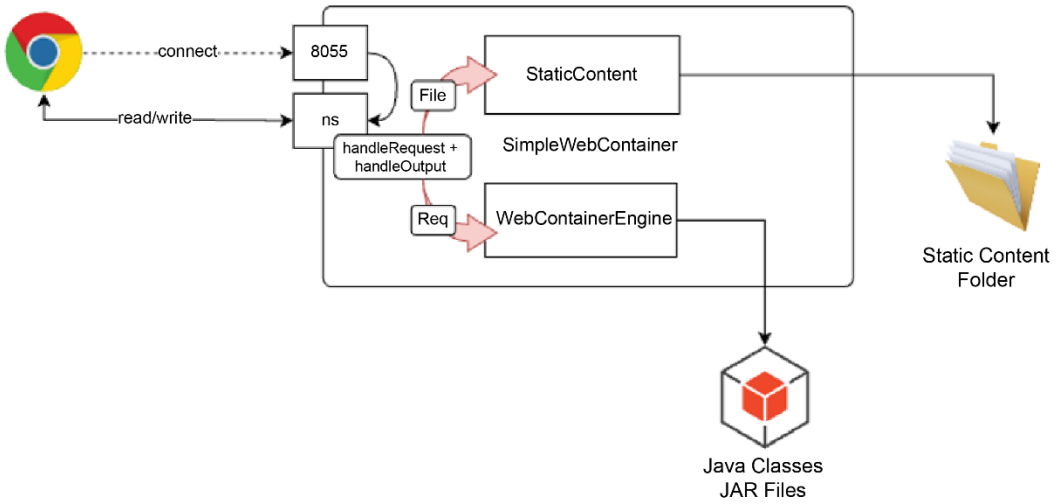


Figure 4.2: SimpleWebContainer Internal Architecture

Let's take a closer look at the illustrated figure. Our SimpleWebContainer is started on a standard port (8055 in the illustrated case). From then on, a new socket (ns) is created for each new request. Up to this point, there is nothing new in relation to the SimpleWebServer version that we studied in the previous chapter.

The difference now is in the handling of the request. We now have a base algorithm that allows us to identify whether we are accessing a file or trying to execute a Java class. If the URL indicates the path to a file, our static content module (StaticContent) is triggered to search for its content and assemble the response. If the URL points to a Java class, we use the Java reflection feature to dynamically invoke the execution of the methods.

For those who are not familiar with Java reflection, this is a feature available since Java version 1.1 that allows us to implement the concept of metaprogramming. What does this mean? It means that it is possible to create Java code that can read, manipulate, and interact with any object instantiated from any class definition. With Java reflection, it is possible to extract metadata (attribute declarations, methods, parameters, annotations, etc.), instantiate new objects, invoke methods, and extract return values from methods in a very simple way.

This way, when, for example, a URL indicates a class to be executed, we can simply load this class and execute the respective method according to the corresponding HTTP method. This also paves the way for thinking about an algorithm that, in general, can handle both types of requests.

Below is an idea of the pseudocode to handle all of this. Note that this algorithm simplifies the complex structure of a complete web server.

```
creates a server socket
while(true){
    accepts new connections
reads request data
extract url from request

    if url contains file extension (htm, html, jpg, png, etc)
        redirects request to static content manager
    else if url contains only a path
        redirects request to web container engine
    else
        generates 404 response

}
```

This small algorithm describes in general terms how our SimpleWebContainer can work, managing both static and dynamic content. But how can we execute this dynamic content in our SimpleWebContainer? We need to create a minimal infrastructure so that we can handle `GET` and `POST` requests through our Java classes.

Handling business rules via HTTP

Considering that we will implement rules for our SimpleWebContainer to handle the `GET` and `POST` methods, we can create a hierarchy of classes that will allow us to manage these methods.

However, just executing the methods is not enough. We need to create representations for our requests and responses, so that our class is able to extract the data from the requests (parameters, message body, headers, etc.). In addition, we need to manipulate the response that we are going to issue as well. For example, we can define a response as "text/html" or "application/json." We can define the body of the response as a serialized object, a complete page, or even the binary content of an image.

Therefore, a small class diagram can help us establish the relationship between the request, the response, and the class that will manipulate these objects. Let's see below:

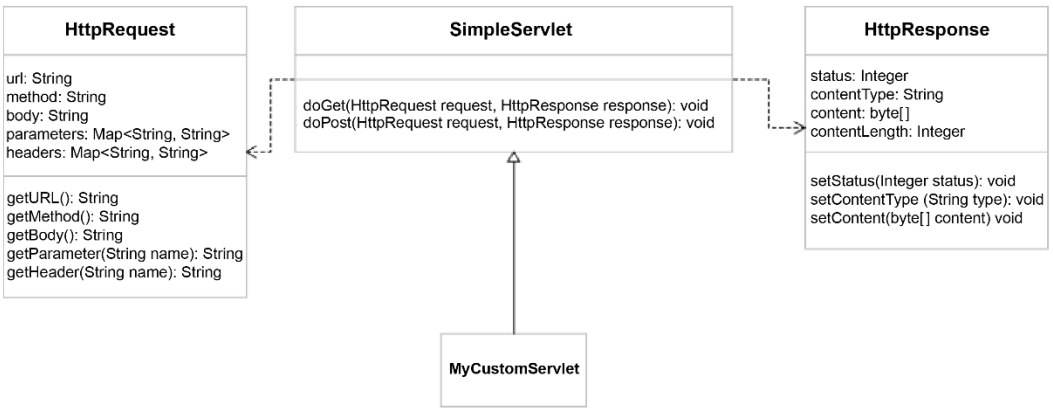


Figure 4.3: SimpleWebContainer Class Diagram for Applications

Basically, what we have here are two classes that represent, respectively, the HTTP request and response (`HttpRequest` and `HttpResponse`). In both, we can see attributes that, as we saw in *Chapter 2*, are characteristics present in each of the entities involved. In the request (HttpRequest), we have the URL, the HTTP method, the message body (body), the URL parameters, the headers, among others. In the response (HttpResponse), we have the status code, the response type (content-type), the content itself (content), which is an array of bytes, among other information.

The class that uses these structures, called `SimpleServlet`, is a class that provides us with two main methods: `doGet` and `doPost`. These methods are executed according to the corresponding HTTP request. In both, we pass the `HttpRequest` and `HttpResponse` objects as parameters so that they can store relevant information both for handling the request and for generating the response.

The idea is that each new class created to handle an HTTP request inherits from the `SimpleServlet` class (as in the example in the image, the `MyCustomServlet` class) and overrides the logic of the corresponding method (`doGet` or `doPost`).

Of course, there is a default behavior for these methods in the `SimpleServlet` class so that it is not necessary to have abstract methods. Even if the subclass does not override both methods (and it is not recommended), the behavior will always be the following:

- status code: 200
- content-type: text/html
- content: "<Request URL> on <ClassName> <Method> Method"

This way, we can ensure that there is a minimum response message for the execution of the method.

However, there is another problem: if we have several URLs, we will consequently have several classes that inherit from SimpleServlet available. How can we differentiate each one and correctly execute the method of the respective class? This can be solved with a specific annotation that we can create.

In this way, we can use Java annotations to solve this impasse. Java annotations are very useful resources for defining metadata for classes, attributes, methods, parameters, or even constructors.

Considering the use of Java annotations, in this case we have the `@ServletDefinition` annotation. We created the annotation for the class scope so that we can even have the same URL for different HTTP methods. In this annotation, we pass the path corresponding to the URL as a parameter. With this additional information, we can create our own classes and associate them with the respective URL, now without any type of conflict. See the following example.

```
@ServletDefinition("/exampleurl")
public class MyCustomServlet extends SimpleServlet{
    public void doGet(HttpRequest request,
                      HttpResponse response){
        // your code
    }

    public void doPost(HttpRequest request,
                      HttpResponse response){
        // your code
    }
}
```

To test our code, we can access the respective servlet (for example, via a browser executing the GET method) by accessing the URL `http://localhost:8055/exampleurl`.

With this defined, we can detail how our SimpleWebContainer locates classes internally.

Considering our basic SimpleWebContainer architecture, we need to define specific paths for static content and dynamic content. Therefore, a very efficient way to segment each content is to have distinct folders. In the case of SimpleWebContainer, we basically have:

- FILES_CONTENT: It is for static content. And from this folder we can have a hierarchy of folders and files.
- WEB_CONTENT: It is to store Java classes (compiled) and JAR files, which allows us to build a web application and host it in SimpleWebContainer. Note that classes on this folder must be previously compiled with references to classes such as SimpleServlet, HttpRequest and HttpResponse.

Now that we have understood all these concepts, how can we bring together all the features seen above into a complete project to evolve a WebServer into an Application Server, even with limited features? Let's explore implementation elements to create a project that supports other developers in creating their own applications and implementing their own business rules.

Implementation issues

How do we put the concepts explained in previous sections into code? We will detail this in the next chapter, but we can already define some requirements.

Our SimpleWebContainer must recognize the HttpRequest, HttpResponse, and SimpleServlet classes, as well as the ServletDefinition annotation. These classes affect the operation and guide the SimpleWebContainer to redirect the processing of the request and the generation of the response.

In addition, we need the applications to be developed independently, that is, in separate projects that do not need or depend directly on the SimpleWebContainer project. How do you do this? Imagine that the SimpleWebContainer is a separate project (or even a binary that is on your machine and will be started independently of your application). The idea is that the classes mentioned above are isolated in a library and made available to anyone who wants to implement an application that will run in this environment.

In this way, the HttpRequest, HttpResponse, SimpleServlet, and ServletDefinition classes are grouped into a specific JAR, which is linked to both the execution of SimpleWebContainer and the definition of the project to be executed on this platform.

Our SimpleWebContainer executes by pointing to specific folders (static and dynamic content), and our classes are copied to these folders. As soon as our SimpleWebServer starts executing, it will already have the folders defined and will know how to execute each of the classes according to the URL requested in the request.

For simplicity in implementation, we will not take into account aspects related to Cross-Origin Resource Sharing (CORS). We assume same-origin requests, so we do not implement CORS headers or preflight.

Summary

In this chapter, we were able to understand the real need for an application server and how we can use a complete specification of an application server to implement a solution that meets some of the requirements of this technology.

We understood how we can reconcile the delivery of static and dynamic content in the same tool so that there are no conflicts and the rules of the HTTP protocol are respected.

We were able to create a simple class model to represent the request, the response, and our class that will serve the requests, as well as annotate it with the desired URL.

And finally, we also learned how we can define specific directories and folders to differentiate static and dynamic content, just like we have in the model in Tomcat, for example.

Handling different contents in different folders will make it easier to implement both the static content module and the Dynamic Content Engine on our Simple Application Server.

5

Deep Dive into Simple Web Container Implementation

How do you implement a **SimpleWebContainer** (**SWC**) from the **WebServer** we already have? We will answer this question throughout this chapter.

Initially, our `SimpleWebServer` is only responsible for delivering static content, that is, reading the bytes of a file and sending them in an HTTP response. The files we handle, in this case, are files that have the following extensions: HTML, HTM, CSS, JS, PNG, JPG, ICO, and TXT. These types of files will only be read and their contents sent according to their media type (Mime Type), so that the browser knows how to display them in the end. Obviously, you can expand this list of files to your needs, such as PDF, ZIP, MP3, AVI, MP4, etc. Regardless of this expansion, the process is the same: search for the file, read its content in bytes, send the response to the browser. If the file is not found, we issue an HTTP 404 error.

In the case of executing Java classes, we first need to have an initial search mechanism for Java classes that allow us to handle requests (Servlets, as seen in the previous chapter). Quite directly and simply, we need to search the project for certain classes (that extend SimpleServlet) and store the necessary information in a table. This table will be our guide for handling HTTP requests.

Therefore, some premises need to be defined so that we can expand our `SimpleWebServer` into our `SimpleWebContainer`.

- We won't handle multiple JVM instances nor use a hot reload strategy
- All apps will be trusted, i.e., we won't check if an application is allowed or not to access our `SimpleWebContainer`
- Our class library to help `SimpleWebContainer` work needs to be available as a set of external libraries (a JAR file or a Maven/Gradle artifact) so that any projects that will be

executed by `SimpleWebContainer` can be created. We will call this set of classes WebContainerLib. The classes are:

  - `SimpleServlet` - the superclass that defines the default behavior of any class that handles HTTP requests
  - `HttpRequest` - to store all data from a request (including headers)
  - `HttpResponse` - to store all data from a response (including status codes, content itself, message size, etc.)
  - Note: the `HttpRequest` and `HttpResponse` classes already exist to handle requests for static content. Now we can not only fetch static content but also receive data from these requests to parameterize the execution of classes that deliver dynamic content, for example.

- Search for classes that extend `SimpleServlet` from the base directory where we will store our classes (or our JAR file). This search should be done using some Depth-First algorithm.
- A data structure to store all the URL paths that will be handled by `SimpleWebContainer`. Using the search service defined in the previous item, instantiate each class and store its metadata in this structure. This will be very useful for dynamically invoking HTTP request handling methods.
- A Request Log module to analyze behavior, displaying messages on standard output (screen)

Other premises can also be considered for a later version, such as:

- `SimpleWebContainer` configuration module to define specific items such as
  - Port number that the server will use to answer requests (different from 8055 that we have already defined)
  - Definition of specific folders for storing static and dynamic content
- More sophisticated logs, using a specific framework for this (such as Log4J for example)

This way, we can start our `SimpleWebContainer` development process.

Basically, our journey in this chapter is

- Specify our library classes, such as classes to represent Http Request and Response
- Specify our main request handling mechanism (our custom "servlet")

- Create a way to explore the whole project finding annotated classes to handle requests (instead of reading XML configuration files)
- Create a complete application that includes a Front End (HTML, CSS, and Javascript) and a BackEnd (Servlets responding to requests) in the same project

Note

As an educational project, SWC will support a very reduced set of features, such as delivering static content in specific file formats, executing Java classes with specific annotations, and supporting GET and POST HTTP methods. SWC will not handle multithreading, cookies, other HTTP methods, or security features.

Creating WebContainerLib classes

For this module, we do not need to make any changes to the `HttpRequest` and `HttpResponse` classes created in *Chapter 3*. We will use their already defined structure to create our base library so that `SimpleWebContainer` works correctly and allows developers to create their own applications.

Let's review the structures of the `HttpRequest` class and the `HttpResponse` class.

```
public class HttpRequest {
    private String httpMethod;
    private String resourcePath;
    private String messageBody;
    private String fileExtension;
    private String destinationHost;
    private Map<String, String> urlParameters;
    private Map<String, String> headers;

    // getters and setters
}
public class HttpResponse {
    private Integer statusCode;
    private String  contentType;
    private Integer contentLength;
    private byte[]  content;
    // status codes
    public static final int OK = 200;
    public static final int BAD_REQUEST = 400;
```

```
    public static final int NOT_FOUND = 404;
    public static final int UNSUPPORTED = 405;
    public static final int INTERNAL_SERVER_ERROR = 500;
    // getters and setters
}
```

Well, from there we can define the basic structure of the `SimpleServlet` class, specifying not only its behavior but also the metadata that will be stored by it.

The `SimpleServlet` class should basically store the path of the URL that it will serve. However, this URL path needs to be defined through some resource in the class declaration. To do this, we will use Java annotations. Since this is specific information, we will also create our own annotation. We call this annotation `ServletDefinition.java`.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ServletDefinition{
    public String path();
}
```

This annotation is declared with just a single method (path) that will receive the URL path and will be very useful for us to achieve greater elegance and simplified verbosity when declaring a class.

Now we can implement our `SimpleServlet` base class, as shown below:

```
public class SimpleServlet {
    private String urlPath;

    public SimpleServlet() {
        Class<?> classDef = this.getClass();
        if (!classDef.isAnnotationPresent(ServletDefinition.class)) {
            throw new RuntimeException("Unable to handle null URL path");
        }
        ServletDefinition servletDef = classDef.getAnnotation
(ServletDefinition.class);
```

```
        this.urlPath = servletDef.path();
    }

    public String getUrlPath() {
        return this.urlPath;
    }

    public void doGet(HttpRequest request, HttpResponse response) {
        response.setContentType("text/plain");
        String responseSTR = "GET: " + request.getResourcePath();
        response.setContent(responseSTR.getBytes());
    }
    public void doPost(HttpRequest request, HttpResponse response) {
        response.setHeader("Content-Type", "text/plain");
        String responseSTR = "POST: " + request.getPath();
        response.setContent(responseSTR.getBytes());
    }
}
```

Let's analyze the class definition.

Initially, we declare a `urlPath` attribute that will be the information we use to determine which path a Servlet will handle. How is this path obtained? Through the definition of the subclasses (the classes you will implement) annotated with `@ServletDefinition`.

In the constructor of this class, we check if it is annotated with `@ServletDefinition`. We use the class definition itself (the first line of the constructor) to get the reference of the instantiated type. Then we check if the annotation exists (this is done by calling the `isAnnotationPresent` method parameterized with the name of the annotation class we want). If the class is not annotated, we throw an exception, interrupting the operation of our `SimpleWebContainer`. Once the class is annotated, the value assigned to the path is retrieved and stored in the attribute.

In addition, both the `doGet` and `doPost` methods have a default behavior to inform the user that they made a request on the given path, but without any logic. This default operation ensures predictable behavior of the base class in case the child classes do not implement these methods.

Now, if we want to implement any class that is executed by `SimpleWebContainer`, we can basically declare it as follows:

```
@ServletDefinition(path = "/hello")
public class MyServlet extends SimpleServlet {
```

```
    @Override
    public void doGet(HttpRequest request, HttpResponse response) {

        // sample code
        response.setStatusCode(HttpResponse.OK);
        response.setContentType("text/html");
        response.setContent("Hello World!");

    }
}
```

A class extending the `SimpleServlet` class must be created and annotated with the path to be served. The method to handle the request should be overridden. An example of a /hello request served by the GET method, which returns a "Hello World" message, has been developed.

Consequently, a single Jar file, potentially a Maven/Gradle artifact in future iterations, can be generated and integrated into any project as a supporting library. This enables developers to incorporate its capabilities for web application development utilizing the `SimpleWebContainer`.

Exploring classes on SimpleWebContainer

Now that we have a project ready to run inside `SimpleWebContainer`, that is, an external project whose classes can be "copied" or "installed" in the respective SWC folders, how can it manage all of this so that it knows exactly which classes to run from the respective URL paths?

Here we need to organize the solution architecture a little. To do so, we will illustrate it in the following image:

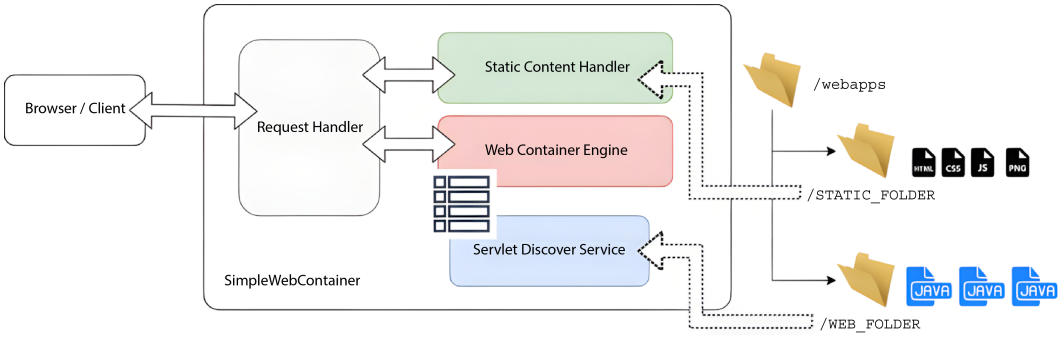


Figure 5.1: Simple web container initial architecture

This architecture is very similar to the one defined in the previous chapter, but here we add a new component: the Servlet Discover Service. This service is responsible for searching the folder

designated for dynamic content (in this case, represented by /webapps/WEB_FOLDER) for all classes that extend the `SimpleServlet` class when the SWC is started.

The strategy adopted for this discovery is simple and uses an on-demand operation.

Initially, a table (a Hash Map) is created where the key is the URL path and the value is the instance of the servlet class. For each request, if the content is not static, we search this table for the class that is annotated with @ServletDefinition and that has the request path. If there is no entry, a search is performed using the Servlet Discover Service in the folder designated for Java classes. Once this class is found, an instance is created and stored in the table. This is useful for serving other requests destined for the same URL, since, if the instance is already stored in the table, searching for the respective class is not necessary.

Therefore, the definition of our servlet table is illustrated below:

```
public class ServletMap{
    public static HashMap<String, SimpleServlet> instances = new HashMap<>();
}
```

To fully understand the Servlet Discover Service, let's first look at a piece of code from the Request Handler to see how the discovery service works.

```
public void handleRequest(Socket socket, HttpRequest request, HttpResponse
response) {
    SimpleServlet servlet;
    String completePath = "/webapps/
STATIC_FOLDER".concat( request.getResourcePath());
    if (Files.exists(Paths.get(completePath))) {
        // send static content
    }
    else {
        servlet = ServletMap.instances.get(request.getResourcePath());
        if (servlet == null) {
            servlet = ServletDiscover.findServlet(request.getUrl());
        }
        if (servlet != null){
            if (request.getHttpMethod().equals("GET"))
                servlet.doGet(request, response);
            else
                servlet.doPost(request, response);
        }
    }
```

```
    else {
        // handle 404 not found response
    }
}
```

In the presented code segment, the request is received through a socket with all attributes populated. Consequently, when a static resource is requested via a URL, the primary validation involves confirming the file's existence. Upon verification, the established process entails reading the file, constructing the response message, and transmitting it to the user, a procedure previously delineated in *Chapter 3*.

Conversely, in the absence of the requested file, the URL is processed as a Servlet. This procedure entails two phases. Initially, a query is executed against the servlet table. If no match is found, discovery via the Discover Service is attempted. Should the Servlet remain unidentified, an HTTP 404 (Not Found) status code is returned.

If the Servlet exists, the request's HTTP method is examined. In the case of a GET method, the servlet's `doGet` method is automatically invoked. For all other methods, the `doPost` method is executed. It should be emphasized that only these two HTTP methods are considered within this implementation, for pedagogical purposes. An excellent strategy for error handling in this case is to return status code 405 (Method Not Allowed), because the user is trying to request another HTTP method that is not implemented.

The mechanism by which the Servlet definition is located is elucidated through an analysis of the `findServlet` method of the `ServletDiscover` class.

```
public class ServletDiscover {

    private static final String DIRECTORY_PATH = "";
    // Adjust to your WEB_FOLDER folder

    public static SimpleServlet findServlet(String urlPath) {
        try {
            Path startPath = Paths.get(DIRECTORY_PATH);
            Servlet instance;

            Files.walkFileTree(startPath, new SimpleFileVisitor<Path>() {

                @Override
                public FileVisitResult visitFile(
                        Path file,
                        BasicFileAttributes attrs) {
```

```
                    if (file.toString().endsWith(".class")) {

                        String classPath = file.toAbsolutePath().toString();
                        String className = convertPathToClassName(classPath);

                        try {
                            Class<?> clazz = Class.forName(className);

                            if (clazz.isAnnotationPresent(
                                    ServletDefinition.class)) {

                                ServletDefinition annotation =

clazz.getAnnotation(ServletDefinition.class);

                                if (annotation.path().equals(urlPath)) {

                                    servlet = (Servlet) clazz
                                            .getDeclaredConstructor()
                                            .newInstance();

                                    ServletMap.instances.put(urlPath, instance);

                                    return FileVisitResult.TERMINATE;
                                }
                            }

                        } catch (Exception e) {
                            // error logging
                        }
                    }

                    return FileVisitResult.CONTINUE;
                }
            });

            return servlet;

        } catch (IOException e) {
            e.printStackTrace();
```

```
        }

        return null;
    }
}
```

This code aims to be very succinct, using the java.nio API for a more efficient search. In this case, we use the walkFileTree method of the Files class, overriding the `visitFile` method to add our search criteria. In this method, we search for all classes and, for each class found, we first evaluate whether it has the `ServletDefinition` annotation in its declaration. This verification is represented by the following snippet:

```
Class<?> clazz = Class.forName(className);
if (clazz.isAnnotationPresent( ServletDefinition.class)) {
...
}
```

If it has this annotation, simply check whether the corresponding annotation has the respective `urlPath` we are looking for as the value in the `path` parameter.

If we find the class with the respective annotation and the value we are looking for, we create an instance, update the Servlet table (for future searches), and finally return the obtained instance so that the request handler can continue. The code snippet corresponding to this is described below.

```
...
ServletDefinition annotation = clazz.getAnnotation(ServletDefinition.class);
if (annotation.path().equals(urlPath)) {
    servlet = (Servlet) clazz
    .getDeclaredConstructor()
    .newInstance();
    // update Servlet Map
    ServletMap.instances.put(urlPath, instance);
    return FileVisitResult.TERMINATE;
}
...
return servlet;
```

An important detail is when our classes are in package structures and these packages also need to be traversed. However, we need to have the base package as a reference for the search (which is

done recursively). For this, we created the `convertPathToClassName` method to convert the package notation to the folder structure (since in practice packages are folders that store Java classes). This method basically finds classes in this package, filters those annotated with the ServletDefinition annotation on the class definition, and returns an instance of it. To avoid creating classes every time, our mechanism stores the created instance in a HashMap to reuse those instances.

```
private static String convertPathToClassName(String classPath) {
    String normalizedPath = classPath.replace("\\", "/");
    int index = normalizedPath.indexOf( DIRECTORY_PATH.replace("\\", "/"));
    if (index == -1) {
        throw new RuntimeException("Path does not contain expected root folder");
    }
    String relativePath
= normalizedPath.substring(index + DIRECTORY_PATH.length() + 1);
    return relativePath.replace("/",".")
    .replace(".class", "");
}
```

This way, it is possible, for each new request, to search for a class definition that corresponds to the Servlet that will handle a given request. The ServletMap works as a temporary storage, avoiding excessive searches for URLs that have already been requested previously and improving the application's performance.

A small example

Let's create a small example of an application that uses `SimpleWebContainer` in action. In this case, we will build a class that extends our `SimpleServlet` to randomly generate a temperature value between 25 and 28 degrees Celsius. Basically, we will have a `/weather` path that returns a JSON object with the following structure:

```
{
    "temperature": 25.35
}
```

In addition, we will create a static HTML page with CSS style and a JavaScript file. This JavaScript file will consume a URL to dynamically fill a section of the page. Therefore, our `SimpleWebContainer` manages and delivers both static and dynamic content.

This static content (HTML page, CSS file, and JS script) is available in the `STATIC_FOLDER` folder, and the servlet that randomly generates the temperature measurement is available in the `WEB_FOLDER` folder.

First, we will define our servlet. Since it will return a specific object, we will initially create our `WeatherDTO` (which is a record that contains the information about the current temperature).

```
public record WeatherDTO(Double temperature) {
    public String toJson() {
        return "{ \"temperature\":" + temperature + "}";
    }
}
```

This record has only one method (toJSON) that converts the structure to a JSON representation because in this example we are not using any object serialization library.

Now we can describe our servlet.

```
import java.util.Random;

@ServletDefinition(path="/weather")
public class WeatherServlet extends SimpleServlet {
    @Override
    public void doGet(HttpRequest request, HttpResponse response){
        // generating random temperature
        Random random = new Random();
        double min = 25.0;
        double max = 28.0;
        double randomValue = min + (max - min) *
        random.nextDouble();
        WeatherDTO dto = new WeatherDTO(randomValue);
        response.setStatus(HttpResponse.OK);
        response.setContentType("application/json");
        response.setContent(dto.toJson().getBytes());
    }
}
```

It is a very simple servlet that, when invoked via a URL in the browser, returns exactly this output.

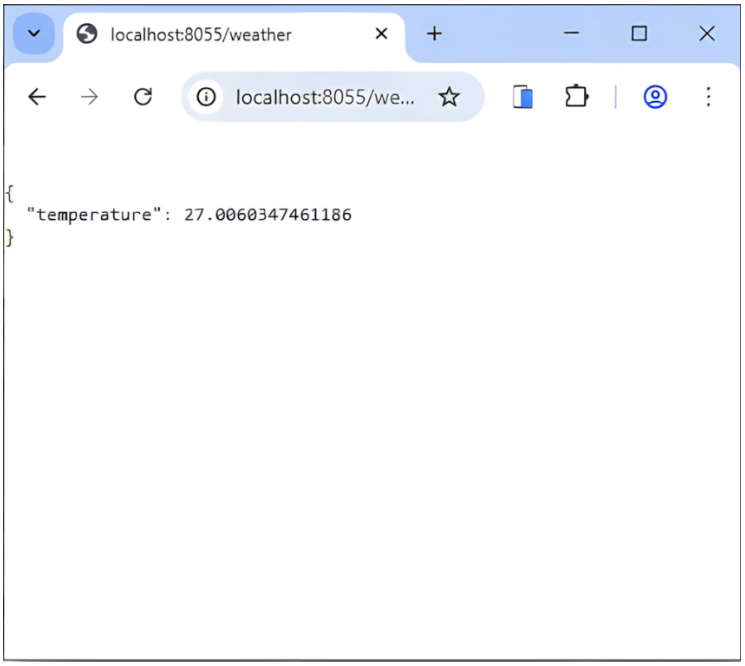


Figure 5.2: Servlet output

So, to have a more user-friendly page experience to display this information, let's go to the HTML, CSS, and JS files.

Here we have the structure of the HTML page.

```
<!DOCTYPE html>
<html lang="pt">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    initial-scale=1.0">
    <title>Weather Forecast</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h1>Current weather</h1>
        <div class="temperature">
            <span id="temp-value">--</span>
            <span class="unit">°C</span>
```

```
        </div>
        <button onclick="fetchTemperature()">Refresh</button>
    </div>
    <script src="script.js">&lt;/script>
</body>
</html>
```

And the STYLE.CSS file content is:

```
body {
    font-family: Arial, sans-serif;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
    background: linear-gradient(to right, #2c3e50, #34495e);
    color: white;
    text-align: center;
}

.container {
    background: rgba(0, 0, 0, 0.7);
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0px 4px 10px rgba(0, 0, 0, 0.5);
    width: 90%;
    max-width: 400px;
}

h1 {
    margin-bottom: 10px;
    font-size: 24px;
}

.temperature {
    font-size: 48px;
    font-weight: bold;
    margin: 20px 0;
}
```

```
.unit {
    font-size: 24px;
}

button {
    padding: 10px 20px;
    border: none;
    background-color: #3498db;
    color: white;
    font-size: 18px;
    border-radius: 5px;
    cursor: pointer;
    transition: background 0.3s;
}

button:hover {
    background-color: #2980b9;
}
```

Finally, SCRIPT.JS file

```
async function fetchTemperature() {
    try {
        const response = await fetch('/weather');
        const data = await response.json();

        if (data.temperature !== undefined) {
            const temp = parseFloat(data.temperature).toFixed(1);
            document.getElementById('temp-value')
            .textContent = temp;
        } else {
            document.getElementById('temp-value').textContent = 'Error';
        }
    } catch (error) {
        console.error("Fetch Error:", error);
        document.getElementById('temp-value').textContent = 'Error';
    }
}
window.onload = fetchTemperature;
```

Testing

To test this example, we basically need to, having our `SimpleWebContainer` running, access the URL (in our case local) `http://localhost:8055/index.html`

And that's it! The page is loaded, with the styling, and the JS file dynamically invokes the Servlet that provides the temperature.

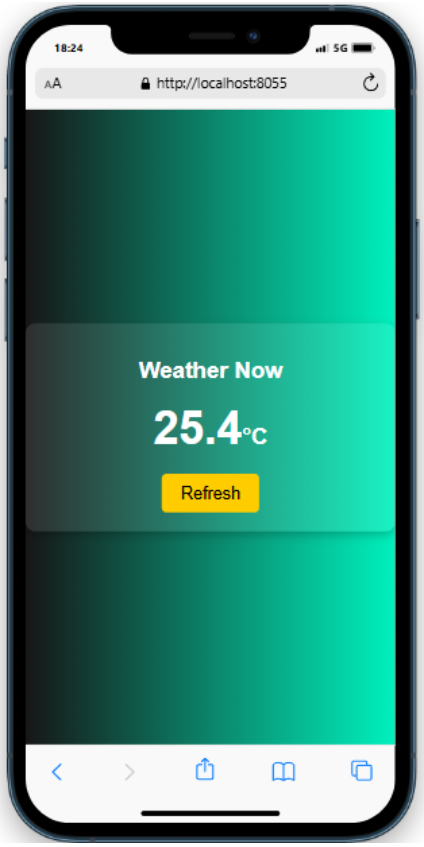


Figure 5.3: HTML page showing dynamic temperature value

Further implementation issues

We can obviously improve the implementation of our `SimpleWebContainer`. Several possibilities can be explored, and since our goal in this book is educational, we can explore several areas for improvement. Let's look at some of them:

- ServletDiscoverService - searching for Servlet classes that are part of the project
- Implementing filters, for example, to manipulate items in the received header, check authorization, create custom responses, etc.
- Improving `SimpleWebContainer` logs (for example, timestamp, HTTP method, path, status code, and others)

With this, we have completed the basic implementation of `SimpleWebContainer`, a minimalist but functional web server capable of delivering both static and dynamic content. The proposed modularity allows for expansion to more robust functionalities similar to production servers. By exploring and implementing the points suggested in the previous section, you will be able to evolve this project to a more advanced level, serving as a solid foundation for custom web applications or even as a teaching tool on the inner workings of web containers.

Summary

In this chapter, we started by specifying the minimal number of classes needed to make a simple application server. The main purpose of this is to explain how we can execute Java classes (with some custom annotations) to handle HTTP requests and send responses. Our `SimpleServlet` class provides methods to handle GET and POST HTTP requests and send responses in different formats. The current limitations of this servlet (GET and POST methods) open a wide range of possibilities, such as implementing other methods and improving content type support. Of course, this Simple Web Container is just a tool for educational purposes. Many of the design decisions we made here are not safe for real environments. But the main goal of this chapter is to show you how static and dynamic content can be handled by a Web Container.

In the next chapter, we will explore more features to expand and improve our Web Container.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

6

Simple Web Container Improvements and Expansions

The previous chapters have progressively presented the construction of a web server from a simple base: the direct manipulation of sockets in Java. This approach, although didactic, allowed us to highlight fundamental aspects of HTTP communication, the role of the web server in the request-response cycle, and the minimum structure required to handle HTTP requests. For example, we can see a full request on our Simple Web Container architecture in *Figure 6.1* below.

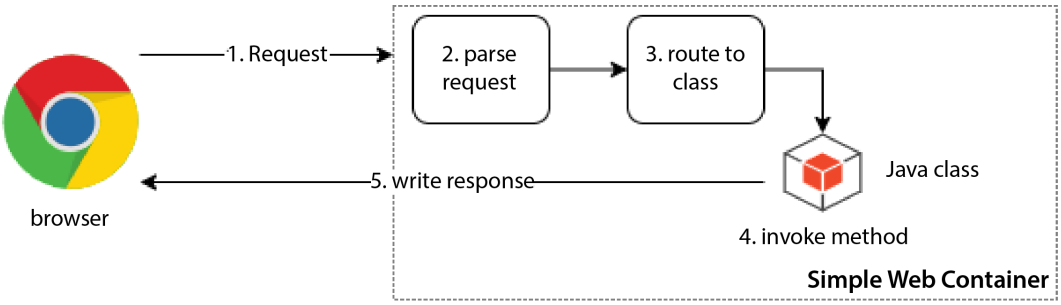


Figure 6.1: How Simple Web Application handles a request

In this chapter, the objective is to expand the functionalities of the Simple Application Server (SAS), highlighting possible improvements in its architecture, discussing its viability as a testing and teaching platform, and outlining ways for it to approach more robust servers, such as Apache Tomcat. To this end, we will analyze aspects such as support for multiple simultaneous requests, session maintenance between requests, support for multiple applications and contexts, monitoring tools, and administration and deployment mechanisms. We will also discuss real-world uses of SAS in specific contexts, especially in resource-constrained environments or educational needs.

Building a web container "from scratch" offers a unique opportunity for in-depth learning about how the web really works. Evolving this container based on best practices and real requirements is the next natural step in this process. For example, if you want to know the full Servlet specification and all features to implement, you can check the documents available at this link:

`https://www.oracle.com/java/technologies/java-servlet-tec.html`

Simple application server review

The Simple Application Server (SAS), built on sockets and standard Java language I/O operations, is a functional and educational web server that implements a basic cycle of listening for connections, reading requests, routing, and responding. Its core structure is organized around a ServerSocket, responsible for waiting for and accepting connections, and a set of auxiliary classes that interpret HTTP requests, prepare responses, and dynamically invoke servlet methods. This blocking I/O approach is intentional to facilitate server implementation.

The current SAS architecture follows a simple flow:

- The server listens on the specified port
- Accepting a connection reads the request and transforms it into a Request object
- Routing is done based on the request path and the invocation of the corresponding method via reflection
- A Response object is filled and sent back to the client

Although functional and useful as a starting point, this architecture has obvious limitations in terms of scalability, security, maintainability, and flexibility. For example, some security risks when implementing a web container could be path traversal, header injection, request smuggling, and others. The remainder of this chapter will cover how to overcome these limitations through incremental, well-defined improvements.

Improvements and expansions

Although **Simple Application Server** (**SAS**) is functional, its implementation is still quite limited for more realistic usage scenarios. This section presents proposals for the evolution of SAS, organized into technical aspects that increase its capacity to serve multiple requests, maintain state between requests, organize multiple contexts, record activities, and facilitate its administrative operation. The idea is to pave the way that would transform SAS into a minimally robust alternative, even in controlled environments or for educational purposes.

Support for multithreading, or even virtual threads

In the current version of SAS, connections are handled sequentially: the server accepts a connection, processes the request, and sends the response before accepting a new connection.

This severely limits the scalability of the server and makes it impractical for any real use with multiple simultaneous clients.

The first natural evolution is the adoption of threads to allow concurrent service of requests. Each accepted connection can be processed by a separate thread.

```
while (true) {
    Socket clientSocket = serverSocket.accept();
    new Thread(() -> handleClient(clientSocket)).start();
}
```

This classic thread-per-request approach, while functional, does not scale well on systems with many simultaneous connections. With the advent of Virtual Threads in Java 21, we now have a lightweight alternative that allows scaling to thousands of connections without the cost of creating native threads:

```
while (true) {
    Socket clientSocket = serverSocket.accept();
    Thread.startVirtualThread(() -> handleClient(clientSocket));
}
```

With this change, SAS is now capable of serving multiple requests simultaneously with much more efficient memory and CPU consumption. This is a fundamental step toward approaching the architecture of servers such as Tomcat, which use optimized thread pools.

One important thing: in this approach, we always create a new thread for each new connection. In production environments, this can be a risk, because without any other control mechanism (for example, limiting the maximum number of threads), it can cause a denial of service. Imagine a huge number of new connections and the Java Virtual Machine creating new threads for each one.

Session maintenance

One of the most critical points in web applications is the ability to maintain state between requests. By default, the HTTP protocol is stateless, that is, each request is independent of the previous ones. However, in real systems, it is essential to store temporary data per user, such as shopping carts, authentication, or preferences.

To achieve this, servers implement the concept of a session, usually based on an identifier persisted via a cookie or URL parameter.

In SAS, we can simulate a session using a ConcurrentHashMap<String, Map<String, Object>>, where each key represents a unique session ID and the value is a map of session attributes. The use of ConcurrentHashMap is to preserve thread safety.

Example of a basic implementation:

```
public class SessionManager {
    private static final Map<String, Map<String, Object>> sessions = new
ConcurrentHashMap<>();

    public static String getOrCreateSessionId(Request req, Response res) {
        String sessionId = req.getCookies().get("SESSION_ID");

        if (sessionId == null || !sessions.containsKey(sessionId)) {
            sessionId = UUID.randomUUID().toString();
            sessions.put(sessionId, new ConcurrentHashMap<>());
            res.addHeader("Set-Cookie", "SESSION_ID=" + sessionId + "; Path=/;
HttpOnly");
        }

        return sessionId;
    }

    public static Map<String, Object> getSession(String sessionId) {
        return sessions.get(sessionId);
    }
}
```

On Request Handler:

```
String sessionId = SessionManager.getOrCreateSessionId(req, res);
Map<String, Object> session = SessionManager.getSession(sessionId);
session.put("lastVisit", LocalDateTime.now());
```

And to retrieve:

```
LocalDateTime last = (LocalDateTime) session.get("lastVisit");
res.write("Your last visit was: " + last);
```

This structure already allows Simple Application Server to behave as a stateful server between requests. This mechanism can be easily extended to support time-to-live mechanisms, periodic cleanup of inactive sessions, and even optional persistence to disk or database. The choice to use a structure such as a Map<String, Object> is to allow developers to store Strings, Objects, and Lists.

Support for multiple applications or contexts

Another significant limitation of Simple Application Server is its inability to handle multiple web applications or contexts. On servers like Tomcat, it is possible to deploy multiple applications, each with its own base path (/app1, /admin, /api, etc.).

To implement this in SAS, it is necessary to structure a context-based routing mechanism. This can be done by mapping path prefixes to sets of servlets or independent application instances. This step must be done at startup by looking for different application contexts in Java files.

```
Map<String, ApplicationContext> contextMap = new HashMap<>();
contextMap.put("/app1", new App1Context());
contextMap.put("/admin", new AdminContext());
```

Upon receiving a request, the Simple Application Server must

1. Extract the path prefix.
2. Check if it matches a known context.
3. Delegate processing to the appropriate context

This model allows applications to be separated logically, and can even have different configurations (such as routes, filters, and loggers) per context. In educational or testing environments, this is extremely useful for simulating scenarios with multiple services or applications running simultaneously. The image below depicts a small pipeline of how new contexts can be implemented in our SimpleApplicationServer.

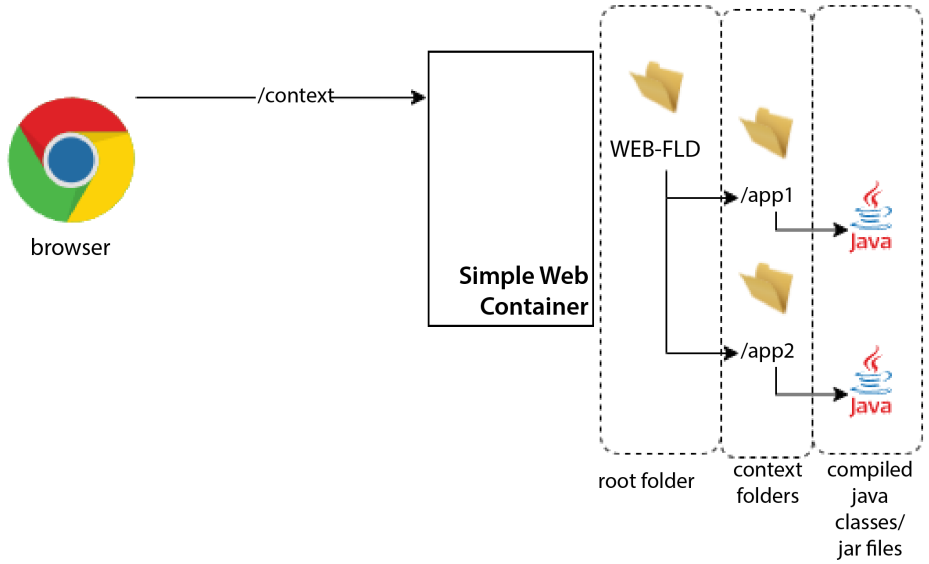


Figure 6.2: Different application contexts organized in folders

Monitoring and logging

Monitoring server behavior is essential for diagnostics, performance analysis, and auditing. Currently, the Simple Application Server can print messages to the console, but it does not have a structured logging system.

A first evolution is the implementation of a Logger class with log levels (INFO, DEBUG, ERROR)

```
public class Logger {
    public static void info(String msg) {
        IO.println("[INFO] " + msg);
    }

    public static void error(String msg, Throwable t) {
        System.err.println("[ERROR] " + msg);
        t.printStackTrace();
    }
}
```

This approach can be refined to write logs to files, generate rotating logs, and even allow viewing on dashboards via a simple web interface. Note: the adoption of console log (such as IO.println or System.err.println) is just for example. A more professional logger tool writes everything in files to keep the records permanent. Of course, logging can have timestamps on each message to order the events. But, for simplification, we just created a very simple logging mechanism to separate responsibilities.

In addition, it is possible to add request counters per second, average response time, and errors per endpoint. This data can be useful even in educational environments for performance analysis.

Tools for administration and deployment

Finally, a minimally complete server must allow dynamic deployment of applications and management of its resources. In real servers, such as Tomcat or Jetty, there are administration consoles for uploading WARs, restarting applications, viewing metrics, etc.

In the context of SAS, it is possible to create a special /admin route (of course with mechanisms to prevent conflicts with other applications), protected by some authentication, that allows:

- Listing loaded contexts
- Viewing active sessions, preserving sensitive data
- Querying recent logs
- Injecting runtime configurations

In addition, it would be possible to allow deployment of applications through .zip files with servlets and static pages. These files would be unpacked into a context directory and immediately activated by the server.

This type of functionality, although more advanced, leads SAS to share professional Application Server features and reinforces its value as an educational and prototyping tool (considering we are simplifying lots of specifications). However, it leads to another feature: class loading and unloading. For example, if you want to keep your Application Server active when you deploy a new version of your application, this feature can be a very hard issue to implement. A very interesting alternative is a hot reload feature, which shuts down and starts up the SAS when a new file or application is added or updated (such as the "Developer Tools" extension in Spring).

Potential uses of simple application server

SAS, even in its basic form or after incorporating the proposed improvements, is a lightweight, transparent, and educational artifact. Unlike robust servers such as Tomcat, which hide part of their complexity under APIs and configurations, SAS is completely open and accessible for experimentation. This makes it particularly valuable in contexts where simplicity, total control,

and ease of customization are more relevant than absolute performance or adherence to corporate standards.

Educational purposes in educational environments

SAS was, from the beginning, conceived as an educational tool. Its source code clearly exposes the fundamentals of how a web server works: sockets, parsing HTTP requests, routing, generating responses, and using annotations, among other concepts essential to web programming.

In Computer Networks, Web Development, or Software Architecture courses, SAS can be used to:

- Demonstrate how the HTTP protocol works "by hand"
- Illustrate the request-response cycle and the concept of static versus dynamic servers
- Explore reflection and rudimentary dependency injection
- Discuss pooling, concurrency, routing, and user sessions

A common exercise, for example, would be to ask students to implement a servlet as follows:

```
@WebRoute(path = "/sum")
public class SumServlet {
    @WebMethod(method = "GET")
    public void doSum(Request req, Response res) {
        int a = Integer.parseInt(req.getParameter("a"));
        int b = Integer.parseInt(req.getParameter("b"));
        res.write("Result: " + (a + b));
    }
}
```

This allows the student to visualize everything from routing to the construction of the HTTP response without relying on external libraries.

Embedded server

The concept of an embedded server has been gaining traction in modern applications, especially microservices, where each application loads its own HTTP server. Embedded servers are very useful to implement microservices, because you can start up a functional REST API (we will discuss it in the next chapters) without any additional or complex configuration. Modern frameworks such as Spring Boot and Quarkus use embedded servers as a minimum infrastructure.

With its simplicity, SAS can be easily embedded in any Java application without the need for deployment in external environments. Simply include the SimpleServer class and invoke it in the application's main() method:

```
public class App {
    public static void main(String[] args) {
        SimpleServer.start(8080);
    }
}
```

And for server shutdown, we can have a shutdown() method that closes the socket and finishes the server. It can be very useful for hot reload applications (such as live servers and developer tools on modern frameworks).

This type of use can be useful for the following:

- Local tools with a web interface
- Internal REST services for legacy systems
- Simplified monitoring dashboards
- Administrative interfaces in standalone software

Given its minimal memory footprint and lack of external dependencies, SAS can be packaged as a single .jar and run with a simple java -jar command.

Use in test environments

Test environments often require a controlled HTTP server with predictable behavior that can be configured in an extremely customized manner. SAS can be more flexible than mock servers (that just have static responses); it can host a minimal application that can handle several use cases, for example.

SAS can be useful for the following:

- Simulating external services (mock servers)
- Integration tests with internal APIs
- Performance tests with load scripts (e.g., Apache JMeter)
- Validation of behavior of internally developed HTTP clients

For example, to simulate an endpoint that always returns a 503 error to test fallback behavior

```
@WebRoute(path = "/unavailable-service")
public class UnavailableService {
    @WebMethod(method = "GET")
```

```
    public void error(Request req, Response res) {
        res.setStatus(503);
        res.write("Service Temporarily Unavailable.");
    }
}
```

Furthermore, since it is not multithreaded by default (unless it is evolved to do so), SAS reduces nondeterminism (you always have the same thread handling a request), which is ideal for unit testing and failure scenarios.

Low-performance devices

Embedded solutions or applications that run on devices with limited resources (such as Raspberry Pi, routers with embedded Linux, or microcontrollers with support for lean JVMs such as GraalVM's SubstrateVM) often cannot use servers such as Tomcat or Jetty, given their memory requirements and startup time.

Since SAS is built directly with sockets and without external libraries, it can run with a minimal amount of heap memory, being highly efficient in:

- Device configuration interfaces (web panel)
- Collection and display of sensor data over HTTP (IoT)
- Local gateways with status interface
- Bridges between local systems and remote APIs

In this scenario, SAS can even be compiled with GraalVM Native Image, eliminating the need for a complete JVM at runtime (some configurations are required for this, but we won't show these details).

```
native-image -jar sas.jar
```

This generates a lightweight binary with very fast boot, ideal for embedded systems.

These examples show that, even without the goal of replacing professional servers, SAS can occupy valuable niches where its accessible, customizable, and lightweight source code makes all the difference.

Summary

Building a web server from raw sockets, as demonstrated throughout this material, proves to be not only feasible but also extremely enriching from a didactic point of view. Simple Application Server (SAS) is a practical synthesis of the study of networks, communication protocols, and the principles of software engineering applied to the web.

This chapter demonstrated that, although SAS was designed in a minimalist way, it has a solid foundation that allows for significant evolutions. The introduction of support for multithreading – or ideally virtual threads – session management, separation of multiple application contexts, and internal monitoring transform this small server into a very versatile artifact. Even though it is not intended to replace servers such as Apache Tomcat, SAS is gaining ground as a lightweight alternative in specific contexts and as an essential learning tool.

We also highlight its applicability in different scenarios:

- Educational environments, where code transparency favors understanding how the web works at a low level
- Embedded environments, where lightness and portability are crucial
- Automated tests or service simulations, where the predictability of SAS surpasses the complexity of complete servers

The practical experience of extending, adapting, and even rewriting parts of SAS can be taken as an academic exercise, challenging students to apply concurrency techniques, add metrics, implement Time-To-Live mechanisms for requests and responses, use component-oriented architecture, apply interface abstraction, and address security.

More important than the server itself is the knowledge acquired along the way. SAS forces us to look behind the curtain of modern frameworks, reminding us that behind every annotation, servlet, controller, or filter lies a web of sockets, input and output streams, and protocols that determine the behavior of the entire web.

In this chapter, we suggested many improvements to SAS features and architecture. If you want to go deeper into web container development, good reference materials are RFC6265 (HTTP State Management), the Servlet Specification, and Project Loom (for virtual threads) for parallelism.

So evolving SAS isn't just about adding functionality. It's about understanding, in depth, every design choice behind widely used enterprise servers and, in doing so, building a solid foundation for the conscious, efficient, and secure development of modern web systems.

Next chapter we will explore the fundamentals of creating a web framework. Web frameworks are essential to create such an abstract layer on this communication and HTTP request/response handling. Currently, with modern frameworks, developers just have to implement application use cases and business rules, keeping away from communication and protocol details.

Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.

`https://packt.link/deep-engineering-newsletter`

7

Why Create a Web Framework

Building **Simple Application Server** (**SAS**) has provided us with a deep understanding of the fundamental mechanics of a web server. From handling sockets to dynamic method invocation via reflection, SAS has taught us how web servers really work under the hood. However, at the same time, it has also shown us that there is a practical limit to the use of hand-crafted solutions in production or more demanding development environments.

While SAS can be scaled, as demonstrated in this chapter, it is not a replacement for robust, well-established servers such as Apache Tomcat, which is maintained by the Apache Software Foundation, implements the official Java Servlet specification, and is widely used by Java applications worldwide. For example, some limitations include the inability to add extensions such as JPA, lack of support for handling timeout requests, and the absence of thread pooling.

By the end of this chapter, you will understand

- The motivation of creating a web framework to simplify all communication and protocol details in web development
- Start defining a minimal framework architecture, delegating responsibilities to specific classes or layers
- Ensure that you only focus on your business logic, leaving all request and response handling to your framework

Why use Tomcat instead of SAS

Apache Tomcat is a Servlet Container, often used as a Java application server that implements the Servlet and **JSP** (**JavaServer Pages**) specifications, enabling the execution of web applications written in Java. It is lightweight, open source, and widely used as a servlet container. Its importance lies in serving as the runtime foundation for many modern frameworks, such as Spring Boot (which can embed Tomcat), by providing a standardized environment for handling

HTTP requests, managing threads, and supporting key Java platform APIs. Thus, even with current frameworks that abstract much of the infrastructure, Tomcat remains essential as a reliable execution engine for Java web applications.

While SAS is educational and transparent, Tomcat offers:

- Implements the Java EE specification
- Native support for filters, listeners, sessions and security
- Automated thread and connection management
- Stable and reliable support for multiple applications
- Monitoring, logging, configuration via XML or annotations. In this case, we can separate responsibilities: application-level configuration uses annotations, while server-level configuration is typically via XML files (server.xml and context.xml)

In addition, Tomcat removes the burden of dealing with problems that have already been solved: connection security, robust routing, compression, session serialization, among others. Of course, some of these features require previous configuration and server tuning.

In enterprise applications, large-scale educational applications, or open Internet systems, using a server like Tomcat is not only recommended – it is typical (and can be considered almost a standard) to ensure performance, security, and scalability.

Understanding servlet: The Tomcat foundation

Tomcat is essentially a servlet container. A servlet is a Java class that implements the javax.servlet.Servlet interface or extends the `HttpServlet` class. The web application logic should be written by overriding methods such as `doGet()` and `doPost()`, and by using the `HttpServletRequest` and `HttpServletResponse` objects. Note: Depending on your Tomcat version, you can find both package names: javax.servlet or jakarta.servlet. This is due to the migration of Java EE (from version 8) to be maintained by the Eclipse Foundation. This migration led to a rename from Java EE to Jakarta EE and, consequently, the package name modification.

Basic Servlet Example

```
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<h1>Hello, World!</h1>");
```

```
    }
}
```

While this model works well for simple cases, it quickly becomes verbose and repetitive. For each endpoint, the developer needs to:

- Handle request parameters directly (getParameter)
- Manually convert types (e.g., from String to int)
- Explicitly manage sessions
- Manually manipulate the HTML or JSON response
- Repeatedly handle exceptions
- Ideally return a status code 400 (bad request) if any request parameter is incorrect
- Write routing logic inside the class itself or via web.xml

This repetition makes development expensive, difficult to maintain, and encourages bad code-organization practices.

Web frameworks: Abstracting infrastructure with Spring and Quarkus

From this complexity, modern frameworks have emerged that abstract the low-level infrastructure of Java web development. Among the most important are the Spring Framework (specifically Spring MVC and Spring Boot) and Quarkus, a more recent alternative focused on cloud-native applications, compatible with GraalVM and with fast startup.

These frameworks encapsulate the use of Tomcat (or another servlet container) and offer several features, such as the following:

- routing via annotations (`@GetMapping`, `@PostMapping`)
- automatic injection of request parameters
- facilitating their processing, automatic serialization of Java objects into JSON, avoiding the need to perform the conversion or search for libraries for this
- simplified application lifecycle, since the framework itself is already responsible for instantiating the objects
- inversion of control and dependency injection, facilitating integration between layers and separating the need for the developer to manage resources
- built-in support for testing, monitoring, security, etc.

To get an idea of the simplification, the same functionality obtained in the previous example can be described in a much simpler way using just a few lines of code. Take a look at the following example with Spring Boot:

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello World!";
    }
}
```

This allows the developer to focus on business logic, rather than on technical infrastructure. The server configuration, routing, parameter parsing, and even some kind of exception handling are handled by the framework (Spring Framework offers the ControllerAdvice annotation to intercept exceptions and write some logic to handle them).

In the case of Quarkus, the same endpoint could be implemented like this (using JAX-RS annotations):

```
@Path("/hello")
public class HelloResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "Hello World!";
    }
}
```

Spring uses Tomcat (or Undertow, Jetty, Netty, depending on the configuration) as the underlying server, while Quarkus is typically based on Vert.x – but the developer rarely interacts with it directly. Thus, the entire Servlet infrastructure is hidden from the developer, as the framework itself is responsible for intercepting the request (in its own servlet), obtaining possible parameters, and redirecting execution to the respective class and method.

In the next section, we will explain how requests are received and handled in the Spring Framework.

Understanding internally how a request works in the Spring framework

One of the key milestones in the transition from hand-built HTTP servers to robust, production-ready web frameworks is the introduction of a central request orchestrator – in the case of Spring MVC, the DispatcherServlet. While the **Simple Application Server** (**SAS**) build provided a valuable foundation in sockets, input streams, reflection, and custom routing, the migration to Spring reveals a structured, extensible, and highly professional ecosystem.

Spring MVC is based on the Front Controller pattern, where all HTTP requests pass through a single servlet – the DispatcherServlet – responsible for delegating each request to the appropriate controller, executing the application logic, and assembling the response. In Spring Boot, the DispatcherServlet is automatically registered at application startup and mapped to intercept all requests sent to the / root path.

Request Interception and Handler Mapping

When an HTTP request hits a Spring Boot application, it is first handled by the built-in servlet container – typically Apache Tomcat, although it can also be Jetty or Undertow. The container forwards the request to the DispatcherServlet. From there, Spring takes over control.

During application initialization, Spring performs a component scan on pre-configured packages. This process identifies beans annotated with `@Controller`, `@RestController`, and other web-related annotations. For each controller class found, Spring uses reflection to inspect its methods and detect mapping annotations such as `@GetMapping`, `@PostMapping`, `@RequestMapping`, and others.

Each method annotated with these annotations is registered in a central internal repository called RequestMappingHandlerMapping. This mapping associates:

- The HTTP verb (GET, POST, etc.)
- the URI pattern (e.g., /products/{id})
- and the Java method of the controller (encapsulated in a HandlerMethod object)

Internally, Spring uses data structures optimized for fast path lookup, including prefix structures (tries) and path matching algorithms, which allow it to quickly locate the target method for any incoming request.

Consider the following example controller:

```
@RestController
@RequestMapping("/products")
```

```
public class ProductController {

    @GetMapping("/{id}")
    public Product getProductById(@PathVariable Long id) {
        return productService.findById(id);
    }

    @PostMapping
    public Product createProduct(@RequestBody ProductDTO dto) {
        return productService.save(dto);
    }
}
```

At startup, Spring registers two routes:

- GET /products/{id} -> method getProductById
- POST /products -> method createProduct

Each route is stored in a RequestMappingInfo object, which includes metadata such as required headers, content types, and path variables. The corresponding Java method is encapsulated in a HandlerMethod, which contains the controller instance, the method reference, and a description of its parameters.

Request Dispatching and Handler Adapters

Once the DispatcherServlet receives a request, it queries the mapping repository to determine which HandlerMethod matches the request path and the HTTP method used. If a matching method is found, the DispatcherServlet uses a HandlerAdapter to invoke the method via reflection.

Before that, however, Spring needs to resolve the method parameters, and this is where argument resolvers come in.

For example, if a method parameter is annotated with `@PathVariable`, Spring extracts the value directly from the URI. If it is annotated with `@RequestParam`, the value comes from the query string. If it is annotated with `@RequestBody`, Spring interprets the request body as JSON and deserializes it into a Java object. To perform this deserialization, Spring uses the HttpMessageConverter abstraction. In the case of application/json requests, the default implementation is MappingJackson2HttpMessageConverter, which in turn uses the Jackson library.

Consider the following DTO and a request sent to the server:

```
public class ProductDTO {
    private String name;
    private BigDecimal price;

    // getters and setters
}
```

Request

```
POST /products HTTP/1.1
Content-Type: application/json

{
    "name": "Bluetooth Speaker",
    "price": 99.99
}
```

Spring detects that the method expects a ProductDTO object annotated with `@RequestBody`. It then uses Jackson's ObjectMapper to convert the JSON into an instance of the expected class. Jackson uses reflection to:

- inspect the class structure
- find the default constructor or record definition
- invoke setters or assign directly to fields
- apply custom deserialization rules when annotated
- The resulting object is passed as an argument to the controller method

Invoking Controller Logic and Generating the Response

After the controller method is invoked and returns a result (usually a domain object or DTO), Spring needs to construct the HTTP response.

If the method returns an object and the controller is annotated with `@RestController`, Spring automatically assumes that this value should be serialized as JSON. Again, it uses the HttpMessageConverter – this time in reverse, for serialization.

Using the `MappingJackson2HttpMessageConverter`, the returned object is transformed into a JSON string using Jackson's ObjectMapper. The generated content is then written to the OutputStream of the HTTP response, and the Content-Type header is set to application/json.

Example of a returned object:

```
public class Product {
    private Long id;
    private String name;
    private BigDecimal price;
}
```

HTTP response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
    "id": 42,
    "name": "Bluetooth Speaker",
    "price": 99.99
}
```

Behind the scenes, Jackson provides fine-grained control over the serialization and deserialization process. Developers can perform the following actions:

- customize field names with `@JsonProperty`
- ignore fields with `@JsonIgnore`
- format dates with `@JsonFormat`
- handle inheritance using `@JsonTypeInfo`

In addition, Spring allows you to configure a custom ObjectMapper through beans, providing global control over naming strategies, field visibility, additional modules (such as Java 8 time API support), and Hibernate integration.

Lifecycle and extensibility

Every step – from route mapping to method invocation to response generation – is modular and extensible. The developer can:

- Add interceptors that execute logic before or after the method
- Register global exception handlers with `@ControllerAdvice`
- Insert filters for authentication, CORS, logging
- Override or add new HttpMessageConverters (e.g. for XML or Protobuf)

This layered architecture, centered on the DispatcherServlet, completely decouples routing logic from business logic, and both from data serialization. The result is a powerful and highly maintainable framework that abstracts away the complexity of the HTTP protocol without restricting customization.

Now that we know how to handle requests and responses, it's time to start our web framework development.

Building our own web framework

Understanding the internal structure of Spring in depth, as well as building a handcrafted HTTP server like SAS, puts us in a privileged position to go one step further: developing our own Java web framework, inspired by modern architectures, but leaner, more educational, and under our total control.

In the next chapters, we will begin building this framework, using an embedded Tomcat as a base and gradually incorporating essential features found in corporate frameworks such as Spring Boot or Quarkus. The idea is to build something functional, modular, and educational, which allows us to understand, test, and expand important concepts of modern web development.

Below, we present the main features that will guide the construction of the framework.

Use of embedded Tomcat

Instead of relying on an external servlet container installed on the system, our framework will run with embedded Tomcat. This approach, widely used in microservices and standalone applications, offers numerous advantages:

- It does not require prior installation of servers
- it makes it easier to package the application as a single .jar
- allows you to start and stop the server programmatically
- enables deployment in cloud-native or restricted environments

Tomcat will be responsible for handling the HTTP protocol, connections, thread control, and initial request mapping. Our code will focus on the upper layers – mainly the routing, control, and injection logic. In other words, Tomcat will map all requests to a single Servlet – our Simple Dispatcher Servlet – that will forward those requests to classes and methods that will handle these requests focused on business logic.

Handling GET and POST Methods

The first feature we will implement is the automatic routing of annotated methods, using an approach similar to what Spring does with `@GetMapping` and `@PostMapping`.

We will allow the developer to annotate methods in Java classes as follows

```
@WebRoute(path = "/products")
public class ProductController {

    @WebMethod(method = HttpMethod.GET)
    public Product list() {
         return new Product("Mouse", 49.99);
    }

    @WebMethod(method = HttpMethod.POST)
    public void create(@RequestBodyObject Product product) {
        repository.save(product);
    }
}
```

From these annotations, the framework will scan the application classes, register the methods and their respective HTTP paths and methods, and dynamically invoke the correct methods based on the received request.

Object Serialization and Deserialization

To support communication in JSON, it will be necessary to implement (or integrate) a robust serialization and deserialization mechanism. We will use Jackson to interpret the request body and convert it into Java objects (in the case of POSTs), and also to automatically serialize objects returned as a response (in GETs, for example).

With this, the framework will allow the developer to focus exclusively on business logic, while the transport of data in JSON is handled transparently, as in the following examples:

```
// Request body
{
    "name": "Keyboard",
    "price": 199.90
}

// Response body
{
    "id": 123,
    "name": "Keyboard",
    "price": 199.90
}
```

This transformation layer will be based on Jackson's ObjectMapper, using modern conventions and allowing for future customization.

Dependency injection (DI) based on the Jakarta specification

Finally, one of the main structural goals of our framework will be to implement a basic dependency injection (DI) container, inspired by the Jakarta Contexts and Dependency Injection (CDI) specification. Obviously, CDI is a huge specification and covers several features. In our simple framework, we will create singleton instances to inject into class attributes.

The proposal is to allow classes and services to be annotated with, for example, `@Inject`, `@Component`, `@Service`, and for the framework to be responsible for:

- instantiating objects in a controlled manner using Java reflection
- resolving their dependencies recursively
- maintaining the coherence of the objects' life cycle
- enabling injection by field initially

This functionality will give us a solid foundation for building decoupled, scalable, and testable services, bringing the framework even closer to professional solutions.

At the end of the process, we will have created not just a functional server, but a complete mini-web framework, 100% written by us, understood in all its internal details, and potentially extensible for real use.

Summary

Building the Simple Application Server provided us with valuable insight into the ins and outs of a web server. We covered the basics of sockets, HTTP protocol manipulation, reflection, manual routing, and response composition. We then moved on to understanding the architecture of modern servers, with a focus on Apache Tomcat and the central role of the DispatcherServlet in Spring MVC, and how frameworks like Spring and Quarkus organize all this complexity for the developer.

But this is not the end. True understanding is only achieved when we are able to apply the concepts learned to concrete solutions. That is exactly what we propose from this point on: developing our own web framework, combining the simplicity of the tools we built with the robustness of the standards we studied.

In the next chapters, we will guide you step by step through the implementation of a modular, lightweight, and didactic framework, using embedded Tomcat, routing via annotations, automatic serialization with Jackson, and dependency injection inspired by the Jakarta CDI specification. This experience will allow us to consolidate all the knowledge we have acquired so far and will open doors to more ambitious projects in the Java web universe.

Building a framework is not just a technical activity – it is a journey of deep learning. And this journey is just beginning.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

8

Web Frameworks – The Necessary Abstraction for Professional Development

Throughout the previous chapters, we progressively and didactically built a journey that began with the fundamentals of network communication, moving on to the structure of protocols and the implementation of a basic web server using Java sockets. We then moved on to the creation of SimpleWebServer, capable of delivering static content. From there, we evolved to **Simple Application Server** (**SAS**), including support for executing dynamic code with Java classes via reflection. In summary:

- We started developing a simple web server to handle only HTTP GET and POST requests
- We produced with SimpleWebServer some minimal static output MIME format, such as HTML, CSS, JS, and some image types (PNG and JPEG)
- Evolved SimpleWebServer to a SimpleApplicationServer delivering static and dynamic content, running Java classes to produce output
    - In this case, a specific API library was necessary. Classes should follow a predefined standard to be able to run on SAS.

This journey allowed us to understand, step by step, how servers actually work behind the abstractions of modern frameworks. We patiently and thoroughly built a simple yet functional web container. We discussed its limitations and evolved its architecture to support multiple contexts, sessions, monitoring, and even administrative routes. In the previous chapter, we analyzed the fundamental role of modern web frameworks – especially Spring and Quarkus – and how they leverage servers like Apache Tomcat to deliver a powerful, modular, and productive

framework. These frameworks not only hide low-level complexity but also offer a complete ecosystem for scalable, secure, and maintainable web applications.

In this chapter, we'll take the next logical step: we'll begin building our own web framework, inspired by modern solutions but with complete transparency and instructional control. To do this, we'll use an embedded Tomcat and apply techniques such as routing via annotations, reflection for metadata extraction, and an initial dependency injection mechanism.

By the end of this chapter, you will understand a simple framework architecture aimed at hiding all request handling complexity. You will be able to create an abstraction layer for HTTP requests and responses and provide your application with functionalities to implement business rules.

Before exploring the framework's architecture itself, it's essential to understand the concept of "framework," especially in the context of web development.

So, what is a framework?

Generally speaking, a framework can be understood as a reusable structure that offers a ready-made foundation for application development. It defines an architecture, provides ready-made components (business logic encapsulated into classes and methods, converters, parsers, and so on), and establishes conventions so that the developer can focus on what really matters: the application logic. In this case, the main difference between a simple class and a component is that the component lifecycle is maintained by the framework (usually using a singleton instance to minimize resource allocation) to offer reusable objects.

Unlike a simple library, which is used on demand and where control lies with the developer, a framework inverts this control. It "calls" the developer's code at the appropriate time – a characteristic known as inversion of control (IoC). This means that the framework determines the flow of execution, and the developer only needs to implement the specific points defined by the framework.

There is a difference between inversion of control and dependency injection: Inversion of control means that your class must run using a "contract," i.e., your class references a Java interface and is not concerned about the implementation. Dependency injection is the mechanism used by the framework to find the best implementation for this interface, manage the implementation's life cycle, and inject the implementation into the interface invocation.

Web frameworks

In the context of web development, frameworks are responsible for handling common and repetitive aspects of the request-response cycle. This includes the following:

Receiving and routing HTTP requests, as this simplifies request handling and allows the developer to focus solely on business logic

- Converting data between formats (JSON, XML), avoiding the arduous work of serialization and deserialization. In this case, we will use an external library (Google GSON)
- Executing logic in controllers, without the hassle of data conversions for each execution
- Generating and sending the response to the client, allowing the delivery of various response types, from text/plain to HTML, XML, or JSON

Modern web frameworks, such as Spring and Quarkus, go further: they support security, testing, database integration, serialization, filters, interceptors, dependency injection, and even transactional control. In our case, our Simple Web Framework will implement only a subset of these features: dependency injection based on the JakartaEE specification. The main purpose is to show how an interface implementation is discovered, instantiated, and injected into a reference.

The great advantage of using a framework is that it drastically reduces the amount of repetitive code required, standardizes the application structure, ensures greater reliability in the application infrastructure, and accelerates development time. However, to use a framework effectively, you need to understand how it works internally.

And that's precisely the purpose of this book.

What do Spring and Quarkus abstract from the developer?

In previous chapters, we observed how, in an application built directly on the Java Servlet specification, the developer must deal with details such as manually converting request parameters, creating objects for each request and managing their lifecycle (for simplification, object lifecycle will be handled as a singleton instance), writing routing logic that requires parsing the request path and HTTP method, manually generating HTML or JSON responses, directly manipulating sessions and headers, and explicitly handling exceptions.

These tasks not only consume time but also make the code more verbose and error-prone. For example, you might forget to set the status code, omit the content-type, return an incompatible response body, and more.

The role of the Spring Framework

The Spring Framework, especially in versions that use Spring Boot and Spring MVC, was created to eliminate this overhead. It encapsulates the use of the server (Tomcat or other) and offers a declarative structure based on annotations. Routing is done with `@GetMapping` and

`@PostMapping`, parameters are automatically converted, and the JSON response is generated based on the returned Java objects. Additionally, Spring automatically manages the lifecycle of application components with a dependency injection (DI) container, which allows objects to be injected wherever needed, without the developer having to worry about instances, lifecycles, or scope.

And Quarkus?

Quarkus follows a similar philosophy, but with a focus on cloud-native applications (applications that are exclusively designed to run in a cloud environment). Its startup is extremely fast, its memory consumption is reduced, and it is compatible with GraalVM (an alternative to the Java Virtual Machine) for native compilation.

Both frameworks share the same principle: hiding the Jakarta EE infrastructure, including the explicit use of Servlets, filters, and request/response objects. All this complexity is handled internally by a DispatcherServlet (in the case of Spring) or a central dispatching component (in the case of Quarkus). Behind the scenes, there's still a Servlet container – usually Tomcat – that handles HTTP connections and forwards them to the framework. The difference is that the developer rarely needs to interact with it directly. Everything is done through annotations, classes, and conventions.

And it's in this sense that not only knowing how to use a framework, but especially understanding its underpinnings, is a task that can make any developer's training stand out.

Why create our own web framework?

At this point in our journey, we've mastered the fundamentals of the web, understood the HTTP protocol, implemented simple servers, and even built an Application Server capable of executing dynamic Java classes. This, in itself, represents a significant technical achievement. However, more important than the code built so far is the deep understanding we've gained along the way. For professional developers, creating your own framework will give you abilities to debug your framework behavior, implement custom extensions, and many other features. You will be able to understand "why" and "how" all mechanisms work internally.

But what now? What's the next step?

Now it's time to transform this knowledge into something even more challenging and revealing: creating our own web framework.

More than writing code: understanding the invisible

When we use established frameworks like Spring or Quarkus, everything seems magical, but it's only engineering. A simple `@GetMapping` annotation causes a URL to be routed automatically. A

parameter annotated with `@RequestBody` is transformed, as if by magic, into a ready-to-use Java object. We return any object, and it is automatically converted to JSON and sent as a response.

But how does it all actually work? This "magic" only ceases to be a mystery when we begin to reconstruct it. And that's precisely our goal with simpleWebFramework: to reveal, layer by layer, what's hidden beneath modern frameworks. Every line of code, every data structure, every annotation, and every reflection call will have an educational purpose: to lead you, the reader, to a true understanding of what happens behind the scenes. For educational purposes, we will intentionally keep the implementation minimal and readable.

A journey guided by three fundamental questions

When creating a web framework from scratch, we begin to ask ourselves questions that might previously seem irrelevant but are essential for any developer who truly wants to master modern backend development:

- How does the server know which class to invoke when receiving an HTTP request
- How can a Java method receive data from the URL, the request body, or even the headers
- How can annotations be used to map URLs, convert data, and control the execution flow of an application

These are questions that frameworks like Spring and Quarkus implicitly answer, as they hide the infrastructure layers from the developer. By building our own framework, we'll answer each of them and understand precisely where the greatest complexities lie and how they're handled by professional solutions.

But first: why leave SAS aside and use Tomcat

Throughout the previous chapters, **SAS** (**Simple Application Server**) has been our laboratory. It's transparent, lightweight, and fully controlled by us. But precisely because of this, it's also limited: it doesn't natively support features like response compression, efficient multiple concurrent connections, structured logging, security support, and other real-world demands of a modern web application.

Using Tomcat Embedded at this point is a strategic choice: we're leaving SAS as a valuable learning experience and relying on a solid, professional foundation – Tomcat – to manage the more robust infrastructure. This frees us to focus on what really matters going forward: framework logic, dynamic routing, metadata extraction, and business logic execution.

In other words, we're not abandoning SAS. We're leaning on it to understand why Tomcat is necessary. At the same time, we're focusing on building what truly differentiates a web framework: the bridge between an HTTP request and application logic, i.e., dispatching, HTTP request handling, and method invocation.

The SimpleWebFramework architecture: a new look at the request-response cycle

With this new goal in mind, we propose the following architecture for our framework

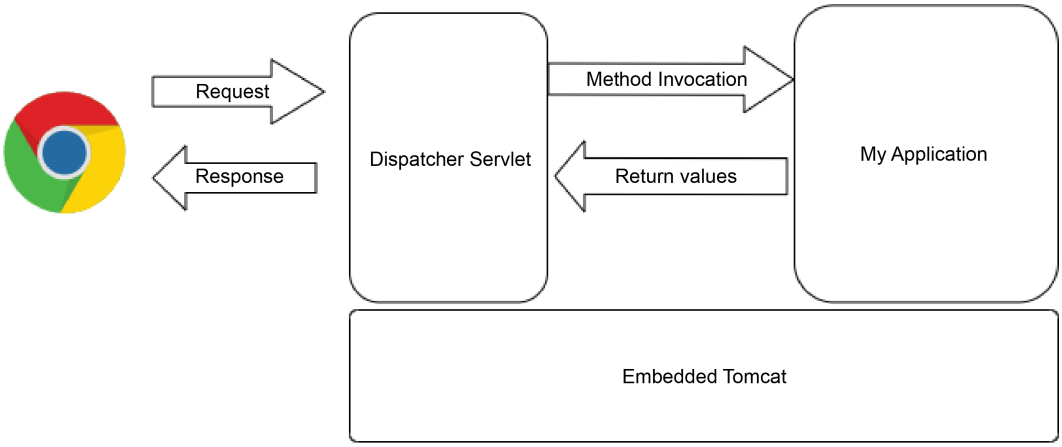


Figure 8.1: Basic architecture of the SimpleWebFramework

Just detailing each of the responsibilities, we have the following:

Browser to HTTP request

The starting point for every interaction in a web application is the browser (or any other HTTP client). It sends a request to a specific URL with a method (GET, POST, etc.), headers, and, optionally, a body.

We've already covered this in depth in previous chapters. Nothing new here, but now we need to consider: how will this request be intercepted and redirected correctly?

Embedded Tomcat: Robust infrastructure, under our control

Unlike SAS, which handled ServerSockets directly, we now use Tomcat embedded. It listens on the HTTP port (8080 by default, but we can customize it using configuration files), manages threads, handles concurrent connections, parses the HTTP protocol, and delivers the already structured request to us through the HttpServletRequest class.

As framework developers, we no longer worry about the protocol. That's Tomcat's responsibility. Our role now is to capture this request and decide what to do with it.

DispatcherServlet: The heart of the framework

Here's the most important component: the DispatcherServlet. It orchestrates all the framework's behavior, hiding the details of request handling and response assembly. From this point on, we have the following desirable scenario:

- Intercept all HTTP requests
- Query the available metadata to identify which class and method should be invoked
- Perform parameter injection (query strings, request body, headers)
- Invoke the corresponding method
- Obtain the return value and convert it into an HTTP response

This second step is precisely the core part of our framework. To do this, we need to create a module capable of extracting the corresponding metadata from our application (whether a class is a controller, or even whether a method of that class will handle a GET or POST request, whether a given method parameter will also be a request parameter, or whether it will come via the request body, and so on).

To facilitate this task, in addition to metaprogramming, we will use the creation of custom annotations (precisely what current frameworks do) to determine the behavior of each class, method, parameter, etc. Among them:

- `@SimpleController` - annotates a class as a controller
- `@SimpleGetRequest` - annotates a method to fulfill a GET request
- `@SimplePostRequest` - annotates a method to fulfill a POST request
- `@SimpleService` - annotates a class to fulfill a service
- `@SimpleInject` - annotates an attribute to inject it as a dependency
- `@SimpleBody` - annotates a parameter to receive it as the body of a request

The metadata extraction module: Understanding the application from annotations

One of the most fascinating aspects of building a web framework is realizing that, unlike a traditional system where the developer needs to manually "register" each feature, here it is the framework itself that discovers and organizes the application's logic based on declarative information – in our case, annotations.

This automated discovery is possible thanks to the use of reflection, a Java language feature that allows you to inspect classes, methods, attributes, and parameters at runtime. With this, the framework can analyze the project's classes, identify which ones are relevant, and build an

internal structure – a veritable map of routes and services – that will guide the entire application's execution.

Thus, our complete framework architecture is illustrated as follows:

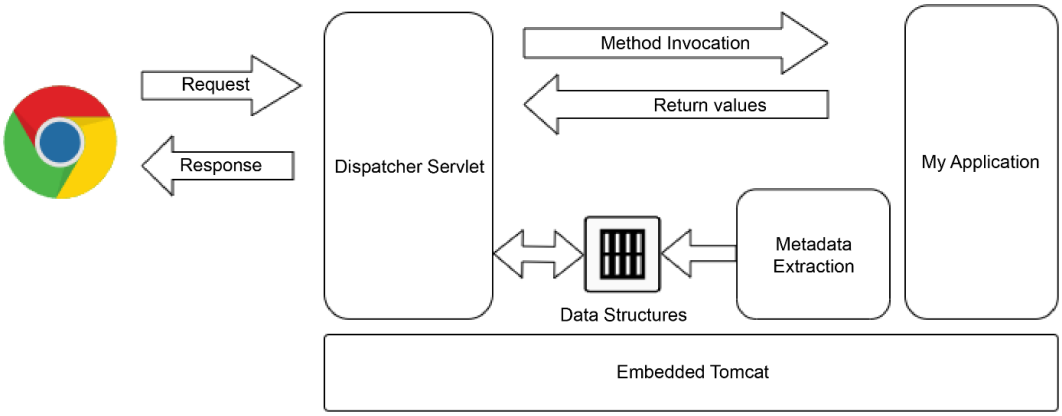


Figure 8.2: Updated simpleWebFramework architecture

Now, each and every element of the application is analyzed by the metadata extraction module, and the respective information is stored in specific data structures. Using Java Reflection, it's possible to extract such metadata from any class and store it in specific data structures (maps or lists). These data structures are useful and essential for the DispatcherServlet to act deterministically, invoking the correct class method even if there are different HTTP operations for the same URL.

Let's understand one by one the annotations that make up this mechanism and how the metadata module uses each of them.

@SimpleController: Identifying controllers

Every web application needs entry points, that is, classes that act as intermediaries between the user request and the application logic. In our framework, these classes are controllers, and they are marked with the `@SimpleController` annotation.

When the metadata module comes into action, it scans the project for classes annotated with `@SimpleController`. Each time it finds one, it understands that this class contains methods that must be exposed to serve HTTP requests. These classes are, therefore, the central blocks of the application's web logic.

@SimpleGetRequest and @SimplePostRequest: Mapping the service methods

Once a controller has been identified, the next step is to understand which methods of that class should be executed upon an HTTP request. Furthermore, you need to determine the type of request and the URL path that should be associated with each method.

This is where the `@SimpleGetRequest` and `@SimplePostRequest` annotations come in. They signal to the framework that a particular method should be executed when a GET or POST request is received, respectively, at a given path.

During the scanning process, the metadata module analyzes all methods of classes annotated with `@SimpleController`, and when it finds these annotations, it records:

1. The request type (GET or POST)
2. The associated URL path
3. The reference to the method, for future execution
4. What parameters the method expects to receive (and how to fill them)

This registration is essential for the DispatcherServlet to correctly redirect each incoming request to its respective handler at run time.

Figure below depicts this full process:

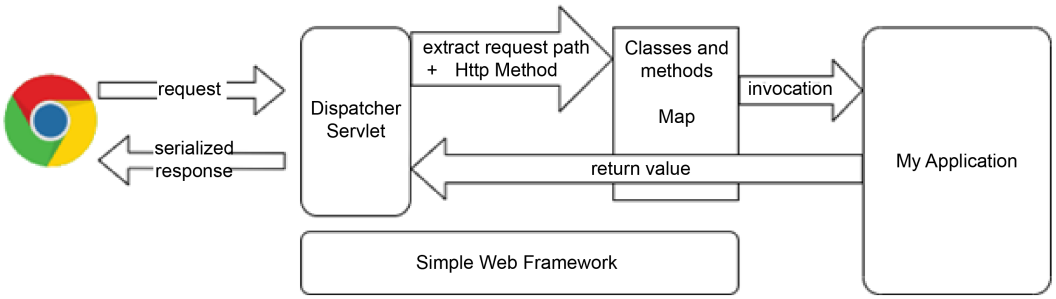


Figure 8.3: Request handling from browser to response

@SimpleBody: Interpreting the request body

Many methods that handle web requests require data that comes in the message body, such as form data or an object sent in JSON.

To let the framework know that a given method parameter should be filled with data from the request body, we use the `@SimpleBody` annotation.

During method parsing, the metadata module identifies which parameters have this annotation and marks this information in the internal structure. This allows the framework to read the request body at runtime, convert it to the appropriate type, and automatically inject this value into the method. A simple error handling mechanism can be implemented, returning a 400 – Bad Request status code as a response if object parsing fails.

This process allows the developer to simply declare a parameter, such as Product product, and the framework automatically handles the conversion without the need to write additional code.

@SimpleService: Organizing reusable components

In addition to controllers, a web application often has services – classes that encapsulate business rules and can be reused by different parts of the system.

In simpleWebFramework, we mark these classes with `@SimpleService`. This tells the framework that the class should be instantiated and managed as a singleton, that is, as a single instance shared by the application. One important thing to clarify is that when we use singleton objects and they keep or update some attributes, they must be thread-safe. Concurrent requests will operate on the same data, and this operation must be synchronized.

These classes don't directly handle HTTP requests, but they are essential for executing actions in controllers.

@SimpleInject: The magic of dependency injection

Once we have services ready and instantiated, the next step is to allow controllers (or even other services) to depend on them without having to manually create them. This is done through the `@SimpleInject` annotation, placed on class attributes.

When the framework encounters a field annotated with `@SimpleInject`, it understands that the field needs to receive an instance of a previously registered service (i.e., annotated with `@SimpleService`). During the initialization process, the framework automatically resolves this dependency.

But, for example, if there are multiple implementations for the same interface, our implementation must notify and not start up. This feature in Spring is solved using the `@Primary`, `@Secondary`, or even `@Qualifier` extra annotations. It can be a new strategy to implement in SimpleWebFramework.

This mechanism is the foundation of dependency injection and ensures that classes are more cohesive, testable, and decoupled – one of the fundamental principles of modern object-oriented design. In our case, we are initially injecting objects using field injection, not constructor injection. We will follow the same strategy used with the `@Autowired` annotation.

Summary

In this chapter, we began a new and important stage in our web development journey. After building, understanding, and expanding our own educational Application Server, we now move on to designing a custom web framework. We explored the central role of frameworks in abstracting technical infrastructure, saw how tools like Spring and Quarkus have transformed the way we develop applications, and delved into the architecture of our simpleWebFramework, analyzing everything from the use of embedded Tomcat to the annotation-based metadata extraction module.

But this construction is just beginning. In the next chapters, we'll explore in detail each of the cogs that will power our framework. We'll see how Tomcat is initialized directly from code, how the DispatcherServlet will be structured to control the entire request-response cycle, and how the metadata extraction module will be developed based on the `@SimpleController`, `@SimpleGetRequest`, `@SimplePostRequest`, `@SimpleInject`, and other annotations. We'll also build the dependency injection mechanism, allowing our classes to collaborate with each other in a decoupled and elegant way. From here on, we won't just be writing code – we'll be revealing the underpinnings of the entire architecture that underpins modern web applications.

9

Detailing the SimpleWebFramework Implementation

Developing your own web framework marks a milestone in the journey of any programmer seeking to deeply understand the inner workings of modern applications. So far, we've built a solid path: we've started with the fundamentals of network communication, implemented a simple HTTP server, evolved into a container capable of dynamically executing classes, and gradually approached the logic of large enterprise frameworks. This chapter represents the culmination of this journey. Our goal now is to understand, line by line, how the SimpleWebFramework is structured and behaves, a didactic implementation inspired by the principles that guide tools like Spring Boot.

The SimpleWebFramework was created with a very clear purpose: to offer a lean, understandable, and functional architecture that abstracts the infrastructure of a modern web server. It allows the developer to focus their efforts on business logic, while the framework itself handles routing requests, identifying controllers, interpreting parameters, and sending appropriate responses. More than that, the framework seeks to transparently demonstrate how each of these mechanisms operates internally – from the moment the server is initialized to the moment the appropriate method is invoked from an HTTP request.

Like Spring Boot, SimpleWebFramework runs autonomously, using an embedded Tomcat. This feature eliminates dependence on external servers and allows the application to function as an independent service, packaged in a single JAR file. This initialization process is driven by the main `SimpleWebFrameworkApplication` class, which is responsible for starting the server, registering the main servlet, and performing an initial scan of the application's classes.

This scan, performed by the `ClassExplorer` class, dynamically identifies all classes annotated with `@SimpleController`, preparing the environment for request dispatch. This is a didactic approximation of the Spring Boot startup process. Using this approach, we can use the embedded server to be our web infrastructure dedicated to our application. In our framework we decided to use Tomcat, but any other embedded Java container (like SimpleApplicationServer developed in previous chapters) can be used.

Once the server is active, every HTTP request is intercepted by the framework's core component – the `SimpleDispatcherServlet`. This servlet acts as the link between the external world and the application logic, analyzing the request, verifying the HTTP method used (GET or POST), interpreting the message body when necessary, and directing processing to the correct method in the corresponding controller. This communication happens automatically, thanks to the use of the `@SimpleGetMethod`, `@SimplePostMethod`, and `@SimpleBody` annotations, which guide the framework on how to handle each request and how to transform the received content into ready-to-use Java objects.

In the following sections, we will explore in detail each of these elements that make up the core of SimpleWebFramework. We will start with the `SimpleWebFrameworkApplication` class, responsible for orchestrating the initialization and registration of the main servlet. Next, we will analyze the functioning of the `SimpleDispatcherServlet` and its routing logic. We will then explore the role of the annotations that guide the framework, as well as how `ClassExplorer` discovers and registers controllers. Finally, we'll see how the `ControllersMap` and `RequestControllerData` data structures maintain the instance lifecycle and ensure that each controller acts as a singleton within the application. By the end of this chapter, we'll have a thorough understanding of the inner workings of SimpleWebFramework and the foundations that underpin its architecture.

By the end of this chapter, you will be able to:

- Delve deeper into understanding the inner workings of a web framework
- Know and understand how to use a Java specification in its implementation
- Have the autonomy to create your own web framework

The main class: SimpleWebFrameworkApplication

In any modern framework, there's always a class responsible for starting the entire application lifecycle. In the case of SimpleWebFramework, this responsibility falls to the `SimpleWebFrameworkApplication` class. It's the starting point, the core that performs the initial configuration, discovery, and registration of the application's components, as well as starting the embedded web server that will serve HTTP requests.

The purpose of this class is to reproduce, in a didactic way, the behavior found in enterprise frameworks like Spring Boot, which uses the embedded server concept – that is, an HTTP server integrated into the project itself, eliminating the need for external configuration and installation. This approach represents a significant departure from the traditional Java EE deployment model, in which applications were packaged in WAR files and deployed to independent servers, such as Tomcat or WildFly. In SimpleWebFramework, as in Spring Boot, the server is part of the application itself and is started programmatically, allowing for complete packaging in a single executable JAR file.

The embedded tomcat dependency

To enable the HTTP server to be embedded in the application, it is necessary to include the Tomcat Embedded dependency in the project, which provides the classes and APIs responsible for instantiating and controlling the web container lifecycle. This dependency is added to the Maven project's `pom.xml` file as follows:

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>10.1.15</version>
</dependency>
```

With this configuration, Maven automatically downloads the Tomcat libraries and includes them in the application's classpath. This way, we can control the server entirely through Java code, creating instances, configuring ports, and registering servlets without relying on XML configuration files or the traditional webapps directory.

Framework initialization

When executed, the `SimpleWebFrameworkApplication` class performs three main steps:

- Discovering the controller classes through the `ClassExplorer` auxiliary class;
- Registering the main servlet, `SimpleDispatcherServlet`, which will be responsible for receiving and processing all HTTP requests;
- Initializing the embedded Tomcat, configuring it to listen for requests on the specified port and binding the servlet to the application's root context.

Below is a simplified version of the code for this class:

```
package simpleframework;

import org.apache.catalina.startup.Tomcat;
```

```
import java.io.File;

public class SimpleWebFrameworkApplication {

    public static void main(String[] args) {
        try {
            // 1. Controllers discovery step
            System.out.println("Starting controllers lookup");
            ClassExplorer.explore();

            // 2. Setup Embedded Tomcat

            Tomcat tomcat = new Tomcat();
            tomcat.setPort(8080);
            // 3. Defining application root context
            String webappDir = new File(".").getAbsolutePath();
            tomcat.addContext("", webappDir);

            // 4. Registering the main Servlet
            tomcat.addServlet("", "dispatcher", new SimpleDispatcherServlet());
            tomcat.addServletMappingDecoded("/*", "dispatcher");

            System.out.println("Server started on port 8080... Ready for
connections");

            System.out.println("Base URL: http://localhost:8080");
            tomcat.getConnector();
            tomcat.start();
            tomcat.getServer().await();

        } catch (Exception e) {
        e.printStackTrace();
        }
    }
}
```

Explaining the code step by step

Executing the main method represents the starting point for any application built with SimpleWebFramework. In the first line, the framework performs a class scan, calling the static `ClassExplorer.explore()` method. This step aims to find all classes annotated with

`@SimpleController` – the classes that represent the application's controllers. The scanning process traverses the project directories, locates the .class files, and uses reflection to identify which ones are correctly annotated. Information about each controller (such as the base path and its request handling methods) is stored in the internal data structures `ControllersMap` and `RequestControllerData`, which will be used later by the core servlet.

After completing the scan, the framework moves on to the second phase: creating the embedded Tomcat server. The new `Tomcat()` code instantiates an object of the `org.apache.catalina.startup.Tomcat` class, which is the entry point for configuring and controlling the HTTP server. Unlike traditional Tomcat, which relies on server.xml files and configuration directories, everything here is defined programmatically.

Next, the `tomcat.setPort(8080)` statement defines the port on which the server will listen for requests. This configuration can be easily changed to allow the application to use any available port by simply changing the value passed as a parameter.

The next step is to define the application's root context – the base directory that Tomcat should consider its execution environment. The line `tomcat.addContext("", new File(".").getAbsolutePath())` creates this context, pointing to the current project directory. The function's first argument ("") indicates that the context will be the server's root path; that is, all requests sent to http://localhost:8080/ will be handled within this context.

The next step is to register the framework's main servlet, `SimpleDispatcherServlet`. This servlet is added to the server with the line:

```
tomcat.addServlet("", "dispatcher", new SimpleDispatcherServlet());
```

The first parameter ("") indicates that the servlet will be registered in the root context, while "dispatcher" is the symbolic name that identifies it internally. Next, the `addServletMappingDecoded("/*", "dispatcher")` method creates the URL mapping, associating all requests (/*) with the newly created servlet. This means that any request sent to the server will be intercepted by the `SimpleDispatcherServlet`, which will then decide which controller and method should be executed.

Finally, the server is effectively started by the combination of `tomcat.getConnector()`, `tomcat.start()` and `tomcat.getServer().await()` calls. The first sets up the Http container associating the port number to be bind, the second initializes the container and begins accepting connections; and the third keeps the process alive, preventing the application from terminating immediately after `main()` is executed. This last statement is crucial, as the HTTP server needs to continue running to serve subsequent requests.

The framework initialization cycle

From an execution flow perspective, the SimpleWebFramework initialization cycle can be summarized in four main stages:

- Class scanning – identifying annotated controllers and methods via reflection
- Server configuration – creating the Tomcat instance and defining the port
- Central servlet registration – associating the `SimpleDispatcherServlet` with the root context and mapping all URLs
- Continuous execution – initializing the server and waiting for new requests

These steps transform a simple Java application into a functional HTTP server, capable of receiving and dispatching requests in a fully automated manner. The great advantage of this architecture is that it allows the developer to avoid worrying about infrastructure details: they simply write annotated classes and methods, and the framework does the rest.

Benefits of embedded tomcat

Using embedded Tomcat brings a series of advantages, both pedagogical, and practical. First, it simplifies the development and deployment process, eliminating the need for WAR packaging and manual configuration of external servers. Second, it allows each application to have its own isolated server instance, reducing the risk of conflicts and facilitating version control. Finally, it promotes a more fluid development experience, especially when combined with modern build tools, which allow for automatic server restarts with each code change.

The ClassExplorer class: locating the Application's controllers

Before we analyze the inner workings of the `SimpleDispatcherServlet`, the heart of the SimpleWebFramework, we need to understand how the framework knows which classes are enabled to receive HTTP requests. The server itself has no knowledge of which classes belong to the application, nor which of them have been designated to act as controllers. Therefore, before dispatching any request, the framework must perform a fundamental step: discovering all controllers annotated with `@SimpleController`.

This task is performed by the `ClassExplorer` class, an essential component in the framework's initialization. This class's role is to traverse the entire project in search of compiled Java classes, check if they are marked with the `@SimpleController` annotation, and, if so, register them in a central data structure that will later be used by the `SimpleDispatcherServlet` for routing and method execution.

This scanning step is the starting point for the framework's "structural awareness": it is through this step that SimpleWebFramework builds its component catalog. While in Spring Boot this process is performed by the component scan mechanism, here it is implemented explicitly, so we can understand every detail of its internal execution.

The role of ClassExplorer in initialization

When the application is executed, the main method of the `SimpleWebFrameworkApplication` class calls the static `ClassExplorer.explore()` method. This call occurs even before the embedded Tomcat is initialized, as it is essential that the framework understand all controllers before beginning to process requests.

The explore() method is responsible for initiating the process of discovering and analyzing the application's classes. In practical terms, it recursively traverses the directories where compiled classes (.class) are stored, applying a depth-first search (DFS) algorithm to ensure that no files are overlooked, even if they are in deeper subpackages.

Class scanning with the Depth-First search (DFS) algorithm

The choice of the Depth-First Search algorithm is no accident. Unlike a superficial search that examines only the top-level directories, DFS allows `ClassExplorer` to explore the entire project directory tree, recursively traversing all subfolders and analyzing each file found. This feature is crucial in real-world projects, where controllers are typically organized into different packages, often subdivided by functional areas, such as `app.controllers.user`, `app.controllers.product`, and so on.

The idea is simple: the method receives a starting directory (usually the base directory of the compiled classes) and, for each item found, does the following:

- If the item is a directory, the method is called again, descending one level in the hierarchy
- If the item is a .class file, it is loaded into memory and inspected via reflection

This behavior is reproduced by the code below, which illustrates how `ClassExplorer` works:

```
package simpleframework;

import java.io.File;
import java.util.Objects;

public class ClassExplorer {

private String BASE_PACKAGE;
```

```
    public static void explore(Class<?> mainClass) {

        BASE_PACKAGE = mainClass.getClass().getPackage().getName();
        String basePath = "target/classes/" + BASE_PACKAGE.replace(".", "/");
        File root = new File(basePath);
        System.out.println("Exploring classes from Base package: " +
BASE_PACKAGE);
        searchRecursively(root, BASE_PACKAGE);
    }

    private static void searchRecursively(File directory, String currentPackage) {
        for (File file : Objects.requireNonNull(directory.listFiles())) {
            if (file.isDirectory()) {
                // Recursive Search - going to next level
                searchRecursively(file, currentPackage + "." + file.getName());
            } else if (file.getName().endsWith(".class")) {
                String className = currentPackage + "." +
file.getName().replace(".class", "");
                try {
                    Class<?> clazz = Class.forName(className);
                    if (clazz.isAnnotationPresent(SimpleController.class)) {
                        System.out.println("Controller found: " + className);
                        ControllersMap.registerController(clazz);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

In this code snippet, the `explore()` function determines the starting point for the scan – typically the application's base package, defined by the `BASE_PACKAGE` constant. The `searchRecursively()` method is the core of the DFS algorithm, responsible for traversing the entire directory structure recursively.

At each iteration, the method checks whether the file is a directory or a compiled class. When it finds a .class file, the method constructs the full class name (including the package) and attempts to load it into the JVM using Class.forName(className). Once loaded, the class can be

dynamically inspected. If it is annotated with `@SimpleController`, it is registered in the global controller map using the `ControllersMap.registerController(clazz)` method.

Using reflection to identify annotations

Reflection allows `ClassExplorer` to dynamically identify annotated classes without relying on any configuration files. The `clazz.isAnnotationPresent(SimpleController.class)` method checks, at runtime, whether the class carries the desired annotation. This operation is extremely powerful, as it transforms Java code into data that can be inspected and manipulated by the program itself.

In frameworks like SimpleWebFramework, this feature is essential for automation. Based on this verification, the framework knows exactly which classes are eligible to act as controllers and can associate them with their respective routes.

The reflection process, at this point, can also be extended to inspect the class's methods and check whether any of them are annotated with `@SimpleGetMethod` or `@SimplePostMethod`. This additional information is encapsulated in objects of the `RequestControllerData` class, which represent a sort of "formal description" of the controller – containing the class name, the Java method to be called, the request type, and the route path.

Directory structure and search behavior

To visualize how `ClassExplorer` works, imagine the following directory structure of an application built on SimpleWebFramework:

```
app/
├── controllers/
│ ├── HomeController.class
│ ├── ProductController.class
│ └── user/
│ └── UserController.class
└── services/
└── ProductService.class
```

When starting the search in the app package, the algorithm traverses each internal folder. Upon entering controllers, it finds the HomeController and ProductController classes, checking both for the presence of the `@SimpleController` annotation. It then recursively descends to the user subfolder, finding UserController.

The final result is a set of three classes registered as controllers. The services folder, although also containing Java classes, will not be included, as its classes lack the specific annotation. Thus,

`ClassExplorer` ensures that only elements intended for handling HTTP requests are registered in the system, ignoring service classes, repositories, or data entities.

Registering the found controllers

Once identified, the annotated classes are sent to the framework's central registry, `ControllersMap`. This component acts as a catalog that associates URL paths with the corresponding controllers. When the `SimpleDispatcherServlet` receives an HTTP request, it consults this map to determine which class should be called and which method should be invoked.

The `ControllersMap.registerController(clazz)` method typically performs three operations:

- Creates a controller instance (usually stored as a singleton to avoid constant re-creation)
- Parses the class's methods, identifying those annotated with `@SimpleGetMethod` or `@SimplePostMethod`
- Stores the set of information (class, methods, annotations, and routes) in an internal data structure

This step is critical because it sets the stage for the subsequent operation of the `SimpleDispatcherServlet`.

Without the initial mapping provided by `ClassExplorer`, the framework would have no way to interpret the incoming requests, nor would it know which method of which class should be invoked to respond to each specific route.

The importance of automated discovery

Within the SimpleWebFramework architecture, `ClassExplorer` represents the first concrete step in routing automation. It replaces manual and verbose configuration with a declarative approach, in which developers simply annotate their classes and methods, leaving the rest to the framework.

This behavior practically illustrates the power of annotations and reflection in modern Java development: code is no longer just a set of instructions, but rather a self-describing and dynamically configurable environment.

Therefore, before we can understand how `SimpleDispatcherServlet` works, we must understand that its entire operation depends entirely on what `ClassExplorer` discovers and records.

It is responsible for building the knowledge base about the controllers available in the application – a veritable route map that will guide the processing of HTTP requests in the framework's subsequent stages.

The SimpleDispatcherServlet: dynamic routing and On-Demand instantiation

After `ClassExplorer` completes its project scan and registers the controllers annotated with `@SimpleController`, the SimpleWebFramework is ready to begin handling HTTP requests.

From this point on, all communication between the client and server is mediated by a single central component: the `SimpleDispatcherServlet`.

This servlet is the entry point for all requests, responsible for interpreting the received URL, identifying the correct controller and method, creating instances on demand, and dynamically invoking the Java code that should respond to the client.

In modern frameworks, such as Spring MVC, this component is known as the Front Controller.

The pattern consists of centralizing the request flow at a single point, delegating to the other parts of the system (the controllers) only the responsibility for implementing the business logic. The `SimpleDispatcherServlet` implements this same concept in a didactic way, revealing the details of how HTTP requests are internally routed and the connection between the web world and Java code.

The relationship between ClassExplorer and DispatcherServlet

Before the `SimpleDispatcherServlet` begins to act, it's important to remember that the `ClassExplorer` has already prepared a fundamental structure: the `ControllersMap`, which serves as a global catalog of all controllers and methods available in the application.

During initialization, the `ClassExplorer` traverses all classes in the project and, for each method annotated with `@SimpleGetMethod` or `@SimplePostMethod`, creates a `RequestControllerData` object.

This class encapsulates all the information necessary for routing a request:

- The URL to be served
- The associated HTTP method (GET or POST)
- The responsible controller class
- And the Java method to be executed

Each of these objects is stored in ControllersMap, a data structure (usually a List<RequestControllerData> or a Map<String, RequestControllerData>) that the servlet will later query.

In other words, when the client accesses a route, such as /products or /users, the framework searches within this map for the match between the URL and the method to be invoked.

ControllersMap and RequestControllerData structure

To understand this process, let's look at a simplified conceptual implementation of these auxiliary classes:

```
public class RequestControllerData {
    private String path;
    private String httpMethod;
    private Method method;
    private Class<?> controllerClass;

    public RequestControllerData(String path, String httpMethod, Method method,
Class<?> controllerClass) {
        this.path = path;
        this.httpMethod = httpMethod;
        this.method = method;
        this.controllerClass = controllerClass;
    }

    public boolean matches(String requestPath, String requestMethod) {
        return
this.path.equalsIgnoreCase(requestPath)&&this.httpMethod.equalsIgnoreCase(requestMethod);
    }

    public Method getMethod() {
        return method;
    }

    public Class<?> getControllerClass() {
        return controllerClass;
    }
}
```

And the central registry of controllers:

```
import java.util.ArrayList;
import java.util.List;

public class ControllersMap {
```

```
    private static final Map<String,RequestControllerData> controllers = new
HashMap<>();

    public static void registerController(RequestControllerData data) {
        controllers.put(data.getPath()+data.getHttpMethod(),data);
    }

    public static RequestControllerData findController(String path, String
method) {
        for (RequestControllerData data : controllers) {
            if (data.matches(path, method)) {
                return data;
            }
        }
        return null;
    }
}
```

These two classes form the core metadata of SimpleWebFramework.

During the discovery phase (performed by `ClassExplorer`), each controller and its annotated methods are transformed into a `RequestControllerData` object and added to the global `ControllersMap.controllers` list.

Thus, when the server is running and the `SimpleDispatcherServlet` receives an HTTP request, it only needs to query this structure to determine which method should be executed.

Internals of the SimpleDispatcherServlet

Below is a simplified code snippet showing the servlet's internals:

```
package simpleframework;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;
import java.lang.reflect.*;
import java.nio.charset.StandardCharsets;
import com.fasterxml.jackson.databind.ObjectMapper;

public class SimpleDispatcherServlet extends HttpServlet {
```

```
    private static Map<String, ControllerData> controllersMap = new HashMap<>();

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {

        String path = req.getRequestURI();
        String method = req.getMethod();

        System.out.println("Received HTTP request: " + method + " " + path);

        // Finds the corresponding controller to handle
        RequestControllerData controllerData
= ControllersMap.findController(path, method);

        if (controllerData == null) {
            resp.setStatus(404);
            resp.getWriter().write("Path not found: " + path);
            return;
        }

        try {
            // Create a controller instance on demand

            Object controllerInstance = controllersMap.get(path + method);

            if (controllerInstance = null) {
                controllerInstance =

controllerData.getControllerClass().getDeclaredConstructor().newInstance();

                controllersMap.add(path + method, controllerInstance);

                // Building method parameters
                Object[] params = buildParameters(req,
controllerData.getMethod());

                // Invoking method by Java Reflection
                Object result =
controllerData.getMethod().invoke(controllerInstance, params);
```

```
                // Serialize JSON response
                ObjectMapper mapper = new ObjectMapper();
                String json = mapper.writeValueAsString(result);

                resp.setContentType("application/json");
                resp.getWriter().write(json);
            }

        } catch (Exception e) {
            e.printStackTrace();
            resp.setStatus(500);
            resp.getWriter().write("Internal Error on request handling.");
        }
    }

    private Object[] buildParameters(HttpServletRequest req, Method method)
throws Exception {

        Parameter[] parameters = method.getParameters();
        Object[] values = new Object[parameters.length];
        ObjectMapper mapper = new ObjectMapper();

        for (int i = 0; i < parameters.length; i++) {

            if (parameters[i].isAnnotationPresent(SimpleBody.class)) {
                // Reading request body (@Isibody annotation)
                String body = new String(
                        req.getInputStream().readAllBytes(),
                        StandardCharsets.UTF_8
                );

                values[i] = mapper.readValue(body, parameters[i].getType());

            } else {
                values[i] = null; // parameters without annotations will be
ignored
            }
        }

        return values;
```

```
    }
}
```

The role of data structures in routing

Note that the servlet's first action is to search for the controller corresponding to the URL and HTTP method:

```
RequestControllerData controllerData = ControllersMap.findController(path,
method);
```

This search runs through the list of controllers registered by `ClassExplorer`.

Each entry (`RequestControllerData`) is compared with the request values – the path and the method (GET or POST).

When a match is found, the servlet has all the information it needs to invoke the correct method of the appropriate class.

This way, the `SimpleDispatcherServlet` does not need to know the application's classes beforehand.

It dynamically queries the catalog prepared by `ClassExplorer`, enabling highly decoupled and extensible behavior: simply add a new class annotated with `@SimpleController` and the framework will automatically recognize it, without requiring any additional manual configuration.

On-Demand creation of controller instances

A particularly interesting aspect of SimpleWebFramework is that controllers are not instantiated in advance.

Unlike complex frameworks that maintain a bean container always active, here, instance creation is on-demand – that is, it occurs only when the corresponding URL is invoked.

This can be seen in this excerpt:

```
Object controllerInstance =
controllerData.getControllerClass().getDeclaredConstructor().newInstance();
```

This line uses reflection to create a new object of the class representing the controller associated with the route.

Thus, if the application has ten controllers, but the client accesses only one of them, only that one will actually be instantiated.

This behavior reduces memory consumption and simplifies the execution model, making the class lifecycle easier to visualize and understand from a teaching perspective.

In future implementations (as will be discussed in the chapter on dependency injection), this on-demand creation could be replaced by more sophisticated management, with singleton instances controlled by a container.

However, for now, the goal is to demonstrate how reflection and dynamic mapping allow the framework to create objects and execute methods without needing to know the code beforehand.

Handling HTTP GET and POST methods

Another essential point is that the `SimpleDispatcherServlet` was designed to handle exclusively the HTTP GET and POST methods.

This choice reflects a pedagogical strategy: to focus on the two most commonly used verbs in RESTful systems and avoid unnecessary complexity.

When querying the `ControllersMap`, the servlet checks whether the corresponding entry uses the same HTTP method as the request (GET or POST).

If a client attempts to send a PUT, DELETE, or PATCH, no match will be found, and the servlet returns a 404 – Route not found error.

This controlled limitation makes the framework's behavior predictable and instructive, reinforcing the direct relationship between HTTP methods and system operations.

Reading the request body with @SimpleBody

For POST requests, the `SimpleDispatcherServlet` supports automatic reading of the message body when the controller method has parameters annotated with `@SimpleBody`.

The helper method buildParameters() reads the entire InputStream associated with the request and converts the received JSON content into a Java object.

For example:

```
@SimplePostMethod(path = "/users")
public User create(@SimpleBody User newUser) {
    return userRepository.save(newUser);
}
```

When the client sends a POST to /users with a body like:

```
{
    "name": "Ana Smith",
```

```
    "email": "ana@example.com"
}
```

The framework reads the request content, converts it to the User class, and automatically injects it into the create() method.

This conversion is performed by the Jackson library through ObjectMapper, as shown in the code.

This process eliminates the need for manual code for JSON parsing, demonstrating how modern frameworks simplify communication between the client and the server.

The `SimpleDispatcherServlet` therefore acts as a universal interpreter between the HTTP protocol and the Java object model.

Consolidated flow

The operation of the `SimpleDispatcherServlet` can be summarized in the following steps:

1. Receives the HTTP request (GET or POST)
2. Query the `ControllersMap` to locate the corresponding `RequestControllerData`
3. Creates an instance of the controller on demand
4. Reads the request body if there is a parameter annotated with `@SimpleBody`
5. Invokes the associated method via reflection
6. Serializes the returned JSON and sends it as a response

This cycle reveals the essence of a modern web framework: transforming HTTP requests into Java method calls automatically, coherently, and extensibly.

Summary

Throughout this chapter, we delved deeply into the internal architecture of SimpleWebFramework, understanding each step that makes a modern web application built entirely in Java possible. This journey began with the `SimpleWebFrameworkApplication` class, responsible for bringing the framework to life. It was this class that launched the embedded Tomcat server, eliminating the need for external servers and allowing the entire system to run autonomously, within a single executable. In this process, the main application also began scanning the project using `ClassExplorer`, a component that laid the foundation for the rest of the execution.

`ClassExplorer` was presented as the true explorer of the application ecosystem. It traversed the compiled class directories using the Depth-First Search algorithm, ensuring that no class was overlooked, regardless of the complexity of the package structure. From this scan, the framework

was able to identify all classes annotated with `@SimpleController`, thus recognizing the entry points that developers had defined to handle HTTP requests. Each identified controller had its methods annotated with `@SimpleGetMethod` or `@SimplePostMethod` analyzed and recorded in a central data structure – the `ControllersMap`. This structure organized the information encapsulated in `RequestControllerData` objects, which represented the link between URLs, HTTP methods, and the Java methods that served them.

Once this discovery phase was complete, the framework's flow moved on to its most important runtime component: the `SimpleDispatcherServlet`. This servlet, registered by Tomcat during initialization, was designed to intercept all requests sent to the server and act as the application's front-end controller. With each request, the servlet queried the `ControllersMap` to find the `RequestControllerData` corresponding to the URL and HTTP method combination. If a match was found, the servlet created, on demand, an instance of the responsible controller and invoked the appropriate method via reflection – a completely dynamic process, without manual configuration.

During this cycle, we highlighted the specific handling of the GET and POST methods, the only ones supported by the framework in its educational version. GET requests, intended for obtaining information, were routed directly to the corresponding method, while POST requests, used for sending data, introduced a new element: the `@SimpleBody` annotation. This annotation allowed the request body, usually in JSON format, to be automatically read, interpreted, and converted into a ready-to-use Java object within the controller method. This way, the framework handled all serialization and deserialization, providing a much cleaner and more intuitive programming interface.

The strength of SimpleWebFramework lies precisely in this integration of reflection, annotations, and automation. Each annotation – `@SimpleController`, `@SimpleGetMethod`, `@SimplePostMethod`, and `@SimpleBody` – acts as a declarative piece that replaces complex configurations. They transform code into an expressive language capable of describing and configuring itself automatically. The use of these annotations, combined with the dynamic mapping provided by `ClassExplorer` and the centralized dispatching of `SimpleDispatcherServlet`, clearly illustrates the principle underlying all great web frameworks: allowing developers to focus exclusively on business logic, while the infrastructure automatically adapts to their declarations.

Thus, the complete flow of SimpleWebFramework – from Tomcat initialization to return in the JSON response – represents much more than a technical exercise. It transparently demonstrates how advanced concepts such as reflection, metaprogramming, dynamic routing, and annotation-based automation integrate to create a powerful and modern development environment. Of course, some features are intentionally missing because the main purpose is educational.

Features such as Path Parameters, validation, and allowed methods can be implemented as new features in next versions.

Understanding this flow means grasping the essence of contemporary frameworks: the ability to convert a set of Java classes into a functional web server, capable of responding to real requests in an agile, organized, and extensible manner.

In the next chapter, we will explore the natural continuation of this architecture: the dependency injection mechanism, responsible for improving the controller lifecycle, promoting instance reuse, and introducing the concept of inversion of control, solidifying SimpleWebFramework as a solid foundation for truly modular and scalable Java applications.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

10

The Emergence of Dependency Injection and Its Application in SimpleWebFramework

Before we begin our studies in Dependency Injection, we need to have a basic understanding of its origins leading up to the current stage.

One problem faced by software engineers was, among others, reconciling the use of classes and resource optimization, as well as the correct search for implementations due to the decoupling of layers. Web systems were emerging rapidly and needed to offer scalability, quick implementation searches, and resource optimization.

In view of these problems, several initiatives began to emerge until what we now call Dependency Injection.

Brief history: the challenge of dependencies in enterprise applications

As software engineering evolved, one concern came to occupy the center of discussions among developers and architects: how to maintain the code of a large application modular, and easy to modify without compromising its stability? This issue became especially critical in the context of Java enterprise applications, which, since the late 1990s, began to grow in complexity and demand scalable, secure, and integrated solutions. For example: a controller being instantiated for each request and in this controller a service and a repo instances are created. Imagine the amount of effort (memory management, memory consumption, and Garbage Collector)

For a long time, corporate Java systems were rigidly structured: each component manually created instances of its dependencies, tightly coupling itself to specific implementations. This meant that any change in a component – for example, replacing a persistence class, a messaging service, or an authentication module – required cascading changes in various parts of the code. This lack of decoupling made the development cycle slow and error-prone.

The concept of Inversion of Control (IoC) emerged precisely as an answer to this problem. Instead of letting each class create its own dependencies, the idea was to invert the creation flow: the framework or container executing the application controls the dependencies. This inversion paved the way for the more specific concept of Dependency Injection (DI), which became the foundation of practically all modern frameworks – including those we will study and build in this book. So, DI is one technique under IoC. Other techniques such as Service Discovery will not be covered in this book.

But before we understand how to apply dependency injection in our SimpleWebFramework, it is essential to understand how this idea was born, matured, and consolidated in the Java ecosystem. And, for this, it is impossible not to revisit one of the pillars of the platform's history: the Enterprise JavaBeans (EJBs).

The birth of enterprise JavaBeans (EJB)

In the late 1990s, Java was already widely recognized as a promising language for corporate applications. Its portability, combined with the ability to execute on robust servers, made it ideal for developing enterprise systems that needed to handle thousands of users and large volumes of transactions.

It was in this context that Sun Microsystems released the Enterprise JavaBeans (EJB) specification – a component of the J2EE (Java 2 Enterprise Edition) platform – with the aim of simplifying the development of distributed, transactional, and secure applications. EJB was conceived to solve a set of common problems at the time: transaction control, concurrency, security, and data persistence, among others.

EJBs introduced a revolutionary concept for the time: components managed by a container. This meant that the developer no longer had to worry about infrastructure details (like connection control or thread synchronization). The container was responsible for creating, instantiating, initializing, and managing the lifecycle of the components, allowing the programmer to concentrate only on the business logic.

In essence, EJB was one of the first major mainstream implementations of the Inversion of Control concept in the Java ecosystem. By handing over the responsibility for object creation and management to the container, the developer started to deal with a highly decoupled application, where dependencies were resolved automatically.

The types of enterprise JavaBeans

With the maturation of the specification, the EJB model began to offer different types of components, each aimed at a specific purpose. Among them, four classic categories stand out:

Session beans

Session Beans were the most common and served to implement the application's business logic. They were invoked by clients (like Servlets, desktop applications, or other EJBs) and existed as long as there was an active interaction with the client.

Within this category, two main variants emerged:

- **Stateless Session Beans (SLSB):** do not maintain state between invocations. Each request is independent, which makes them ideal for quick operations, such as calculations or simple queries. For example, a freight calculation service can be implemented as a Stateless Bean, as the value of each calculation does not depend on a previous request.
- **Stateful Session Beans (SFSB):** maintain state between invocations by the same client. This means the container needs to associate the bean instance with a specific session. This model was useful for applications that required continuity between interactions, such as shopping carts or authentication flows.

Message-Driven beans (MDBs)

With the growth of messaging-based architectures, the need for components capable of reacting to asynchronous events arose. Message-Driven Beans were created for this purpose. Instead of being invoked directly by a client, they are triggered automatically when a message arrives at a queue (usually via Java Message Service – JMS).

MDBs are a clear example of the application of the Inversion of Control principle: the container monitors the queue, receives the messages, and, when necessary, creates or reuses a bean instance to process them, without the developer having to worry about thread control or listeners.

Entity beans

Entity Beans represented persistent data, usually mapped to database tables. Each instance of an Entity Bean corresponded to a record in a table, and the container was responsible for automatically synchronizing the data between the database and the Java object.

Although Entity Beans (Container-managed or Bean-Managed) were a pioneering attempt to integrate the object-relational paradigm into Java, their complexity and low performance eventually led to the emergence of simpler alternatives, such as JPA (Java Persistence API), which consolidated the concept of entities as simple annotated POJOs (Plain Old Java Objects) managed

by a persistence provider. Bean Managed Persistence leads the user to write SQL statements for each operation on each object. It was hard and less productive. Container Managed Persistence requires no SQL code to be written, but a huge amount of configuration files (XML files) were needed to map each object. JPA brought an elegant and efficient way to map objects to SQL relations using Java Annotations.

The role of the EJB container and XML configuration

One of the central elements of the EJB model was the container, a kind of "controlled execution environment" that managed all aspects of the beans' lifecycle. The container was responsible for:

- Creating and destroying bean instances according to demand
- Automatically injecting dependencies
- Managing transactions and concurrency
- Applying interceptors and filters declared in the deployment descriptor
- Handling automatic failures and rollbacks

However, in the early years of the J2EE platform, this sophistication was accompanied by a major disadvantage: the complexity of the necessary configurations.

For each EJB, it was necessary to declare its behavior in an XML file known as the deployment descriptor (usually called *ejb-jar.xml*). This file described, in detail, the bean name, its type (stateless, stateful, etc.), the interfaces it implemented, the dependencies the container should resolve, and the transaction and security policies.

A simplified example of classic EJB configuration:

XML

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>OrderService</ejb-name>
<ejb-class>com.company.ejb.OrderServiceBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
<assembly-descriptor>
<security-role>
<role-name>manager</role-name>
</security-role>
```

```
</assembly-descriptor>
</ejb-jar>
```

This need for detailed and manual configuration made development with EJBs a bureaucratic and slow process. Small code changes required updates to the XML files, recompilations, and complete system redistributions.

On the other hand, it was precisely this declarative model – where dependencies and behaviors were defined outside the code – that paved the way for the natural evolution of automatic dependency injections and configuration via annotations that we know today.

From XML configuration to annotation: the maturation of IoC

Over time, the Java community realized that the XML model was powerful, but excessively verbose. The developer needed to manually describe what the compiler and runtime itself could deduce. Starting with Java EE 5, launched in 2006, a new paradigm emerged: configuration by annotations. CDI itself arrived on a later JavaEE version, starting on the JavaEE 6 era.

Now, instead of relying on long XML files, the developer could use annotations directly in the code to inform the container how the bean should be managed. Thus, the same previous example could be rewritten much more simply:

Java

```java
@Stateless
public class OrderServiceBean {
    @PersistenceContext
    private EntityManager em;

    public void processOrder(Order order) {
        em.persist(order);
    }
}
```

Here, the container automatically understands that the class is a stateless bean and injects an EntityManager instance into the `em` variable thanks to the `@PersistenceContext` annotation. This model brought three major benefits:

1. Less coupling: classes no longer need to manually create their dependencies
2. Greater legibility: the application's behavior is explicit in the source code itself
3. Greater automation: the container now has total control over the objects' lifecycle

This movement marked the definitive consolidation of Dependency Injection as an essential technique in corporate Java development.

Dependency injection: the power of specification

Dependency Injection (DI) is one of the most widespread concepts in modern software architecture, especially in the Java ecosystem. However, what many developers do not realize is that, in the corporate Java world, it is not just a development technique applied by frameworks like Spring or Quarkus – it is, in fact, a formal specification, called Jakarta Contexts and Dependency Injection (CDI).

CDI is not an exclusive feature of a framework or library. It is an open standard, maintained under the umbrella of the Jakarta EE project, that defines a set of rules, behaviors, and annotations that any dependency injection container must follow to be considered compatible. Just as the HTTP protocol defines how browsers and servers communicate, CDI defines how dependency injection should work within the Java platform, in an interoperable, and predictable way.

This means that any framework that wishes to offer dependency injection – be it Weld (the CDI reference implementation), Quarkus, Micronaut, or even Spring – can adopt the specification and offer compatibility with its fundamental concepts and annotations. Here it's important to note that some frameworks cannot implement the full specification (for example, Micronaut supports the JSR-330 style injection annotations and has its own container semantics). The benefit of this is immense: standardization. A developer who understands how CDI works is capable of applying the same principles in any modern framework, without depending on proprietary resources.

Bean types and structural decoupling

At the heart of the CDI specification is the concept of **Bean Type**. In essence, a bean is any class managed by the container – that is, the container is responsible for instantiating it, maintaining its lifecycle, and injecting it where necessary. The bean type defines under which types this bean can be injected.

This is what guarantees the decoupling between parts of the application: the consuming code does not need to know the concrete implementation of a service, only its abstraction. The container is the one that knows which class corresponds to that abstraction and, therefore, which instance it should provide.

Consider the following example in Java:

```
public interface PaymentProcessor {
    void process(Payment payment);
}
```

```
@ApplicationScoped
public class PaypalProcessor implements PaymentProcessor {
    @Override
    public void process(Payment payment) {
        System.out.println("Processing payment via PayPal");
    }
}
```

Here, the `PaypalProcessor` class is a managed bean. Its bean types include `PaymentProcessor`, `PaypalProcessor`, and `Object`. This means it can be injected into any point of the application that declares one of these types.

A client could simply write:

```
@Inject
PaymentProcessor processor;
```

and the CDI container would automatically inject an instance of `PaypalProcessor`. Note that the consuming code does not know which class is being used. The decoupling is complete – the programmer only declares what they need ("a payment processor"), and the container provides the appropriate instance.

This is the key point of CDI: the inversion of control is institutionalized by the specification. The developer stops instantiating directly (`new PaypalProcessor()`) and only declares dependencies, while the container is responsible for the resolution. This makes the application extensible, testable, and modular.

A more complete example shows the power of bean types. Suppose there is a second implementation:

```
@ApplicationScoped
public class StripeProcessor implements PaymentProcessor {
    @Override
    public void process(Payment payment) {
        System.out.println("Processing payment via Stripe");
    }
}
```

Now there are two classes that implement the same `PaymentProcessor` interface. Both share the same bean type. For the container to know which one to inject, **qualifiers** come into play – annotations that add an "identity stamp" to the bean and the injection.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE,
ElementType.METHOD})
public @interface Paypal {}

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE,
ElementType.METHOD})
public @interface Stripe {}
```

These qualifiers are applied to the classes and dependencies:

```
@Paypal
@ApplicationScoped
public class PaypalProcessor implements PaymentProcessor { ... }

@Stripe
@ApplicationScoped
public class StripeProcessor implements PaymentProcessor { ... }

@Inject
@Paypal
PaymentProcessor processor;
```

Now the container knows exactly which instance to provide. This mechanism is defined by the CDI specification and is widely followed by modern frameworks.

The concept of bean type, therefore, is the link that unites polymorphism, inversion of control, and standardization. It ensures that dependencies are resolved by type and context, without the developer needing to resort to factories, service locators, or reflective code.

Implementations and compatibility: spring and quarkus

Two of the best-known implementations that apply (or are strongly inspired by) the CDI specification are the Spring Framework and Quarkus. Both adopt the philosophy of inversion of control, but do so in slightly different ways, each with its own historical context and ecosystem.

Spring framework

Spring, besides being one of the most popular frameworks in the Java language, emerged before the standardization of CDI and, for this reason, developed its own dependency injection infrastructure, based on proprietary annotations such as `@Autowired`, `@Component`, `@Service`, and `@Repository`.

Over time, however, Spring evolved to adhere to the CDI specification, supporting JSR-330 annotations `@Inject`, `@Qualifier`, `@Named`, and `@Singleton` – all defined in the `jakarta.inject` and `jakarta.enterprise.context` packages.

Today, in a modern Spring application, it is possible to use both the framework's original annotations and those of the specification. The following two examples are equivalent:

```
// Using standard CDI annotations
@Inject
private PaymentProcessor processor;

// Using Spring annotations
@Autowired
private PaymentProcessor processor;
```

Internally, the behavior is the same: Spring searches the `ApplicationContext` for a bean whose type is compatible and injects it at the corresponding point. If there is more than one possible implementation, the framework requires disambiguation, done through `@Qualifier` or `@Primary` – concepts directly inspired by CDI qualifiers.

This compatibility ensures that Spring applications can benefit from a unified conceptual basis with the rest of the Jakarta EE ecosystem. In other words, a developer familiar with CDI will not find it difficult to work with Spring, and vice versa.

Furthermore, Spring Boot introduced automatic bean detection mechanisms through component scanning, a behavior equivalent to the CDI bean discovery process. Thus, both discovery, and dependency resolution follow almost identical principles, although with distinct terminology and implementations.

Quarkus

Quarkus, on the other hand, was born already within the Jakarta EE era and adopts CDI as its backbone. Unlike Spring, Quarkus did not need to create its own annotations or abstractions – it implements the specification directly.

All dependency injection in Quarkus is done using the standard annotations:

```
@ApplicationScoped
public class EmailService {
    public void send(String to, String msg) { ... }
}

@Singleton
public class NotificationManager {
    @Inject
    EmailService emailService;

    public void notifyUser(String user) {
        emailService.send(user, "Notification sent!");
    }
}
```

Here, `@Inject`, `@ApplicationScoped`, and `@Singleton` are all annotations defined by the Jakarta specification. The Quarkus CDI container is a highly optimized implementation called Arc, designed for ultra-fast startup and low memory consumption, without altering the specification's default behavior.

This means that CDI code executed on Quarkus is functionally identical to what would run on a traditional Jakarta EE server (like WildFly, Payara, or TomEE). The same code could even be migrated between these environments with few or no modifications.

This degree of fidelity is what makes Quarkus one of the purest examples of CDI application. Its adherence to the specification is so rigorous that it even supports context extensions, custom scopes, and interceptors defined by the standard.

While Spring was born independent of Jakarta EE and evolved to partial compatibility, Quarkus was conceived entirely within the specification, guaranteeing portability, and predictability between containers.

A common base, multiple expressions

Both Spring and Quarkus demonstrate that CDI has become a common language among Java frameworks. The concept of bean type, type resolution, the use of qualifiers, and context management have become universal practices, even when implemented differently.

- Spring adopts `@Autowired` and @Qualifier("name"), but also understands `@Inject` and `@Named`

- Quarkus exclusively uses the standard annotations `@Inject`, `@ApplicationScoped`, `@Singleton`, `@Named`, and `@Qualifier`
- Both follow the same resolution logic: finding a bean whose type and qualifiers correspond to the injection point

This convergence is the true strength of the specification: standardizing the way dependencies are declared, resolved, and managed, regardless of the container's concrete implementation.

CDI, therefore, is not just another component of the Jakarta EE platform, but one of the conceptual foundations of modern Java development – a model that made possible what frameworks like Spring and Quarkus practice with excellence today.

Implementing dependency injection in SimpleWebFramework

In previous chapters, we explored the natural evolution of our SimpleWebFramework: it started as a simple static content server, evolved into an Application Server capable of executing Java code in response to HTTP requests, and finally, became a web framework capable of automatically mapping controllers and methods through annotations.

Each of these steps brought a new layer of abstraction and automation. But until now, despite the automation in routing and request handling, there was one point that remained manual: the creation and management of dependencies.

Whenever a controller needed to use an auxiliary service (for example, a data repository, an authentication component, or a messaging provider), this dependency still needed to be instantiated manually, within the code itself. This created coupling, made maintenance difficult, and prevented the framework from achieving the level of independence and modularity observed in modern enterprise frameworks.

The introduction of a dependency injection mechanism represents the next step in architectural evolution. From now on, the developer declares what they need – and no longer how it should be created. The framework's container becomes responsible for locating, instantiating, and managing the application's objects.

This is, in essence, the philosophy of Inversion of Control (IoC), formalized in the Jakarta Contexts and Dependency Injection (CDI) specification. And this section focuses precisely on this transition – from manual creation to automated injection.

CDI as a conceptual base and inspiration

Since CDI is a specification, any product or framework that follows these rules can be considered compatible with the CDI model. This is what makes the Jakarta EE ecosystem so powerful and

interoperable: the same code that works on a Payara server can, with minimal changes, work on WildFly, Quarkus, or even a Spring Boot application, as all follow, to a greater or lesser extent, the same conventions.

CDI specifies concepts such as:

- Beans: classes managed by the container
- Bean Types: types under which a bean can be injected (interfaces, superclasses, concrete classes)
- Qualifiers: annotations that differentiate multiple implementations of the same type
- Scopes: object lifecycles (request, session, application, dependent, etc.)
- Contexts: environments that control the visibility and duration of each bean

In our SimpleWebFramework, we do not implement CDI completely – that would be a large-scale project. But we will build a light dependency injection module that follows the same philosophy:

- Automatically locate classes eligible for management
- Map interfaces and their implementations
- Create instances on demand or in advance
- And automatically inject the necessary dependencies into the consuming classes

This is the same principle that guides real frameworks, such as Quarkus, which uses the Arc implementation for CDI, and Spring, which historically created its own IoC container even before the specification was standardized.

Our objective is didactic: to deeply understand the internal mechanisms of a framework, and at the same time respect the formal principles defined by Jakarta EE.

The @SimpleService annotation: defining managed beans

In a real CDI container, any class that has a *bean-defining annotation* (such as `@ApplicationScoped`, `@RequestScoped`, `@Singleton`, or `@Dependent`) is automatically recognized as a managed bean.

Inspired by this concept, the SimpleWebFramework introduces its own marking annotation: `@SimpleService`.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface SimpleService {}
```

This annotation is the way the developer informs the container that this class must be automatically discovered and managed. During the framework's initialization process, the

`ClassExplorer` scans the application packages and identifies all classes annotated with `@SimpleService`.

The behavior is similar to Spring Boot, which identifies classes annotated with `@Service`, `@Component`, or `@Repository`, and Quarkus, which detects classes with `@ApplicationScoped`.

The difference is that, in SimpleWebFramework, there is no external configuration file. The entire process is automatic and reflection-driven, minimally reproducing the behavior of an IoC container.

A practical example:

```
@SimpleService
public class EmailServiceImpl implements EmailService {
    public void send(String to, String msg) {
        System.out.println("Email sent to " + to);
    }
}
```

Here, the `EmailServiceImpl` class is registered as a managed bean. The framework automatically identifies that it implements the `EmailService` interface and adds this association to an internal map of implementations (*ServiceImplementationMap*), as follows:

```
EmailService → EmailServiceImpl
```

This means that, anywhere in the application, when requesting an injection of `EmailService`, the container will know which concrete class must be used.

This strategy respects the concept of bean types defined by the CDI specification: the same bean can be injected through any type it exposes (interface, superclass, or the class itself). This detail ensures total decoupling between the code that consumes a service and the code that implements it.

The role of ClassExplorer in service discovery

The `ClassExplorer`, introduced in the previous chapter, was originally responsible for finding classes annotated with `@SimpleController` and their methods annotated with `@SimpleMapping`. Now, its responsibility increases: besides locating controllers, it also needs to identify and register the application's services.

The execution flow of the `ClassExplorer` for this new stage is as follows:

1. Traverse all application packages
2. Identify classes annotated with `@SimpleService`
3. Check if the class implements any interface
4. Create an association between the interface and the concrete class
5. Register this association in the `ServiceImplementationMap`

This structure is the heart of dependency injection. It stores all available services and allows the container to know, at any moment, which implementation should be used to satisfy an injection.

The simplified code for the process is something similar to:

```
for (Class<?> clazz : allClasses) {
    if (clazz.isAnnotationPresent(SimpleService.class)) {
        for (Class<?> iface : clazz.getInterfaces()) {
            ServiceImplementationMap.register(iface, clazz);
        }
    }
}
```

This step is crucial because it keeps the system abstraction-oriented. The container never injects concrete classes directly – it always injects interfaces, exactly as CDI advocates.

In a real container, this process is known as *bean discovery*, and it is widely configurable. In Spring Boot, for example, *component scan* fulfills the same role, sweeping declared packages in search of components.

In our case, we simplified the process, but maintained its essence: locate, register, and maintain a map of services. Even for simplification, we do not consider multiple implementations for the same interface. It would be necessary to implement an extra annotation such as @Qualifier (or @SimpleQualifier), and it can be an extra improvement for next versions.

The @SimpleInjected annotation: the injection point

Once the services have been identified and registered, the next step is to allow other classes – such as controllers, or even other services – to automatically inject these dependencies.

For this, the SimpleWebFramework defines the `@SimpleInjected` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface SimpleInjected {}
```

This annotation marks the fields that must be automatically filled by the container. During execution, the framework iterates through the loaded classes and identifies the annotated fields. For each field, it consults the `ServiceImplementationMap` in search of a registered implementation for the requested type.

If found, the container checks if an instance of that class already exists in the cache (*DependencyInjectionMap*); otherwise, it creates a new instance via reflection and injects it into the field.

The process is the same concept applied in enterprise frameworks:

- In CDI, the field would be annotated with `@Inject`
- In Spring, with `@Autowired`
- In our framework, with `@SimpleInjected`

An example:

```
@SimpleController
public class NotificationController {
    @SimpleInjected
    private EmailService emailService;

    public void sendWelcome(String user) {
        emailService.send(user, "Welcome to the system!");
    }
}
```

During the initialization cycle, the container detects that the `emailService` field needs to be injected. Upon finding `EmailService -> EmailServiceImpl` in the implementation map, the framework creates (or reuses) the instance and automatically injects it into the controller.

From this moment on, the developer no longer needs to manually create objects. The SimpleWebFramework takes charge of managing all dependencies, bringing into practice the concepts of IoC and DI described by the Jakarta CDI specification.

We know that we can inject an object in both ways: annotated field or constructor parameter. Again, for simplification of implementation, we adopted the annotated field strategy.

The DependencyInjectionMap and the instance lifecycle

The `DependencyInjectionMap` is the structure responsible for storing the already created instances of services. It acts as a *singleton registry*, that is, a global repository that prevents the repeated creation of objects that should exist only once during application execution.

When the framework needs to inject a dependency, it follows this flow:

1. Checks if an instance for that type already exists in the DependencyInjectionMap:
    - If it exists, it returns that instance
    - If it does not exist, it creates a new one and stores it
2. Returns the instance to the injection point

This behavior corresponds to the **application scope** defined in CDI – where each bean has a single live instance throughout the application's execution.

Spring, by default, adopts the same behavior for its singleton beans, initializing them at startup and keeping them in memory.

In SimpleWebFramework, we can choose between two main creation strategies: on demand and at startup, as detailed below.

Instantiation strategies

Instantiation on demand

The *on-demand* approach consists of creating objects only when they are needed. If a dependency is never used, it is simply never instantiated.

This strategy's main advantage is the efficient use of resources, especially in applications that load various modules but use only a fraction of them. It also reduces the server startup time, as it postpones object creation until they are actually requested.

In SimpleWebFramework, on-demand instantiation is the default behavior: upon finding a dependency annotated with `@SimpleInjected`, the container checks if an instance already exists in the `DependencyInjectionMap`. If not, it creates the instance at that moment and stores it for future injections.

This mechanism is conceptually equivalent to what Quarkus does in development or lazy mode, where injection is deferred until the code that depends on the bean is executed. In Quarkus, this strategy directly contributes to the extremely short startup time and low memory consumption, which are the framework's hallmarks.

Instantiation at startup

The other strategy is *eager initialization*, also called *instantiation at startup*. In this model, all services are created right at the beginning of the framework's execution, generally during the server initialization phase.

This approach is adopted, for example, by the Spring Framework, which creates all singleton beans at the time the `ApplicationContext` is constructed. The main benefit is immediate

availability: when the server finishes starting, all dependencies are already ready, and the response time to the first request is practically zero.

Furthermore, configuration errors (such as unresolved dependencies or missing implementations) are detected during initialization, avoiding runtime failures.

For SimpleWebFramework, instantiation at startup can be easily activated by modifying the `ClassExplorer` flow. After identifying and registering all services, the framework can traverse the `ServiceImplementationMap` and create the corresponding instances in advance, storing them in the `DependencyInjectionMap`.

With this, the application starts with all services loaded and ready for use, following the same model as Spring Boot.

Each approach has its advantages:

- On-demand instantiation reduces consumption and speeds up initialization
- Eager instantiation ensures stability and lower latency

The choice between one and the other may depend on the type of application and its environment constraints.

Integration with the DispatcherServlet lifecycle

Dependency injection does not act in isolation; it is deeply integrated into the execution flow of the `DispatcherServlet`, the framework's central component responsible for orchestrating the handling of HTTP requests.

When a request arrives, the `DispatcherServlet` performs the following steps:

1. Route Identification: locates the controller and method corresponding to the requested URL
2. Controller Instance Creation (or Retrieval): the framework checks if the controller has already been instantiated; otherwise, it creates a new one
3. Dependency Injection: before method execution, the container iterates through all controller fields annotated with @SimpleInjected and performs the injection, following the mechanism described previously
4. Business Logic Execution: with all dependencies resolved, the method is invoked, processing the request and generating the response

This flow ensures that the developer can write simple and clean controllers, without infrastructure code.

For example:

```
@SimpleController
public class UserController {
    @SimpleInjected
    private UserService service;

    public HttpResponse listUsers(HttpRequest request) {
        List<User> users = service.findAll();
        return HttpResponse.ok(users);
    }
}
```

The execution of this request involves a complete chain of abstractions:

- The ClassExplorer had already identified UserService as @SimpleService
- The `ServiceImplementationMap` mapped UserService -> UserServiceImpl
- The DispatcherServlet creates the UserController
- The container automatically injects UserServiceImpl into the service field
- The method is executed normally

This behavior is identical to that observed in frameworks like Spring MVC and Quarkus RESTEasy, where the programmer writes only the application logic, and the container takes care of the rest.

The role of decoupling and modular architecture

The most visible result of this implementation is the total decoupling between the application's modules. With the automated injection model, the developer starts thinking in terms of contracts, not implementations.

This separation is what allows for:

- Replacing one implementation with another without changing the client code
- Facilitating unit tests, using mocks or stubs instead of real classes
- Reusing components in different contexts (for example, the same service can be used in REST applications and batch processing routines)
- Dividing the application into independent modules, each with its own development cycle

This type of architecture is the foundation of any scalable system. In real frameworks, such as Spring, modularity, and dependency injection are the factors that allow the creation of entire ecosystems – security, persistence, messaging, caching – all integrated transparently.

In SimpleWebFramework, this modularity begins to materialize. The developer no longer needs to "know" where or how each service is created. They just need to declare it. The container resolves, injects, and maintains the objects' lifecycle, in a controlled, and predictable way.

Summary

By reaching this point of the chapter, it is possible to observe how the SimpleWebFramework evolved from a didactic server to a mini application container with enterprise-level features. Implementing the dependency injection mechanism meant much more than just eliminating the need for `new`.

It was the consolidation of a series of fundamental concepts of modern software engineering:

- Decoupling: each application module knows only its logical dependencies (interfaces), not the concrete implementations
- Modularity: services can be replaced, tested, and versioned independently
- Lifecycle Management: the container takes responsibility for creating, storing, and reusing instances
- Extensibility: new services can be added without modifying existing code
- Standardization: by following the model of the Jakarta CDI specification, the framework adheres to a set of internationally recognized best practices

More than learning to program an injection container, the developer who follows this implementation learns to read and interpret a Java specification – a formal technical document, such as Jakarta Contexts and Dependency Injection 4.0, which defines behaviors, terminologies, and contracts.

Knowing how to navigate this type of specification is a valuable skill, which distinguishes the developer who only uses tools from the one who understands their fundamentals. This understanding is what allows one to create frameworks, contribute to open source projects, and understand how enterprise technologies truly work internally.

With this, the SimpleWebFramework reaches a new level: it stops being just an academic exercise and comes to represent, in miniature, the architecture of the great modern Java frameworks. A didactic, but fully functional model, which demonstrates in practice how the power of abstraction and standardization transforms complexity into elegance.

In the next chapter, we will make a full example of using the SimpleWebFramework, considering a new application using it as the infrastructure to be deployed as a microservice.

11

Creating the First Application with SimpleWebFramework

In the previous chapters, we have covered a long journey of evolution for our web server: we started from communication via sockets and the direct manipulation of HTTP requests, created the SimpleWebServer, evolved to the SimpleWebContainer, and finally understood the architecture of modern frameworks, such as Spring Boot and Quarkus.

These frameworks brought to the Java ecosystem the ability to abstract all the infrastructure of a server, automating processes such as routing, dependency injection, and data serialization. However, understanding each layer of this process is what truly transforms the programmer into a complete developer – capable of building or extending any Java web application consciously.

From this point on, we move from the building stage to the utilization of our own framework: SimpleWebFramework. We will create a full application using SimpleWebFramework as our communication infrastructure.

It represents the practical synthesis of everything we have learned so far. It is a minimalist, didactic, and functional framework that offers a solid base for the development of modern web applications without requiring the typical complexity of corporate frameworks.

Just as with Spring Boot or Quarkus, the SimpleWebFramework is also distributed as an independent Java artifact, which can be packaged and installed in the local Maven repository. In this case, we used Java 25 as the standard compiler and Virtual Machine, and Maven 3.9.

Once we have developed the framework project, we just need to install it locally so it can be used in any new application.

The process is simple and follows the Maven standard

```
mvn clean install
```

This command compiles the project, packages the generated artifact (e.g., `simplewebframework-1.0.jar`), and installs it in the local Maven repository (~/.m2/repository), making it available as a dependency for any other Java projects.

Creating the MyWebTestApplication project

With the framework installed, we can now create the first application based on it. This application will serve as our demonstration project, showing how the `SimpleWebFramework` manages controllers, services, and entities automatically and transparently.

The initial project structure follows the Maven standard, as presented below.

```
mywebtestapplication/
├── src/
│   └── main/
│       ├── java/
│       │   └── com/myapp/
│       │       ├── MyWebTestApplication.java
│       │       ├── controllers/
│       │       │   └── HelloController.java
│       │       ├── services/
│       │       │   ├── IService.java
│       │       │   └── IServiceImplementation.java
│       │       └── model/
│       │           └── Product.java
│       └── resources/
│
└── pom.xml
```

In the `pom.xml` file, we include `SimpleWebFramework` as a dependency:

```
<dependencies>
<dependency>
<groupId>com.simpleframework</groupId>
<artifactId>simplewebframework</artifactId>
<version>1.0</version>
```

```
</dependency>
</dependencies>
```

With this, all the functionalities of the framework – annotations, dependency injection, route mapping, and embedded server – become available to the project.

The main class: MyWebTestApplication.java

The entry point of the application is the `MyWebTestApplication.java` file.

```
package com.myapp;

public class MyWebTestApplication {
    public static void main(String[] args) {
        SimpleWebFramework.run(MyWebTestApplication.class);
    }
}
```

This simplicity is intentional and reflects one of the fundamental principles of the `SimpleWebFramework`: doing the essential without syntactic overhead. Unlike frameworks such as Spring Boot and Quarkus, which use annotations like `@SpringBootApplication` or `@QuarkusMain` to mark the application's entry point, the `SimpleWebFramework` adopts a more direct approach.

It is not necessary to explicitly indicate, through an annotation, which class initializes the system, as the existence of the main method (in this case still using the syntactic pattern prior to Java 25) invokes `SimpleWebFramework.run`, passing the definition of the application class itself as a parameter. This is because, through this definition, it is possible to capture the package it belongs to and, subsequently, perform the discovery of all the components of this application (controllers and services).

Nevertheless, there is a notable structural similarity between applications written with these different frameworks. Observe the examples below:

Spring Boot:

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Quarkus:

```
@QuarkusMain
public class DemoApplication {
    public static void main(String... args) {
        Quarkus.run(DemoApplication.class, args);
    }
}
```

SimpleWebFramework:

```
public class MyWebTestApplication {
    public static void main(String[] args) {
        SimpleWebFramework.run(MyWebTestApplication.class);
    }
}
```

In all three cases, the purpose is the same: to start the application's lifecycle. The main method is the entry point that triggers the embedded server initialization mechanism, and, unlike Quarkus and SpringBoot, as mentioned earlier, the SimpleWebFramework adopts the strategy of on-demand instantiation. Once started, no service or controller class is instantiated unless a request is made.

By observing the initialization log, some messages might suggest the existence of instances, but they were included with the simple objective of showing that the framework's class explorer only found classes in the project definition. The instantiations have not been made yet.

```
  __ _                 _      __    __     _
 / _(_)_ __ ___  _ __ | | ___/ / /\ \ \___| |__
 \ \| | '_ ` _ \| '_ \| |/ _ \ \/  \/ / _ \ '_ \
 _\ \ | | | | | | |_) | |  __/\  /\  /  __/ |_) |
 \__/_|_| |_| |_| .__/|_|\___| \/  \/ \___|_.__/
                |_|
2025-11-12 09:13:33 Embeded Web Container         :Starting MyWebTestApplication
2025-11-12 09:13:33 Metadata Explorer             :Found a Controller
br.com.isiflix.webtestapp.controller.HelloController
2025-11-12 09:13:33 Metadata Explorer             :Found a Service Implmentation

br.com.isiflix.webtestapp.service.IServiceImplementation
2025-11-12 09:13:33 Metadata Explorer             :    Class implements
br.com.isiflix.webtestapp.service.IService
```

```
2025-11-12 09:13:33 Embeded Web Container         :Web Container started on port
8080
2025-11-12 09:13:33 Embeded Web Container         :MyWebTestApplication started in
0.405 seconds
```

Controllers and route mapping in SimpleWebFramework

One of the most striking features of a modern web framework is the ability to transform common Java methods into HTTP endpoints directly, intuitively, and declaratively. The `SimpleWebFramework` was designed precisely to offer this experience in a lightweight and transparent manner, without requiring complex configuration layers.

Just annotate a class as a controller and, within it, indicate which methods should respond to GET or POST requests. The framework takes care of everything else: route mapping, request handling, method execution, and automatic response formatting in JSON.

The `HelloController.java` file is the most didactic example of this philosophy. It demonstrates that, with very few lines of code, it is possible to create a complete set of REST endpoints capable of both returning simple messages and sending and receiving Java objects.

Below is the updated controller code.

```
package com.myapp.controllers;
// import section

@SimpleController
public class HelloController {

    @SimpleGetMapping("/hello")
    public String sayHello() {
        return "Hello from SimpleWebFramework!";
    }

    @SimpleGetMapping("/product")
    public Product getProduct() {
        return new Product(1, "Keyboard", 199.90);
    }

    @SimplePostMapping("/product")
    public Product createProduct(@SimpleBody Product product) {
```

```
        product.setId(2);
        return product;
    }
}
```

The structure of this controller is minimalist and highly expressive.

The `@SimpleController` annotation identifies the class as a web component, allowing the framework's Class Explorer to detect it automatically at startup.

Each method annotated with `@SimpleGetMapping` or `@SimplePostMapping` is interpreted as an HTTP route, with the path attribute responsible for defining the URL that triggers that method.

The `sayHello()` method is the most basic example of an endpoint. Upon receiving a GET request on the /hello route, the framework automatically invokes this method and transforms the returned text into a standardized JSON string.

Even if the return is a simple string, the `SimpleWebFramework` formats the response to ensure compatibility with front-end applications and integration tools. The result, therefore, is delivered to the client as:

```
"Hello from SimpleWebFramework"
```

There is no need to manually configure serialization libraries or define HTTP headers – the framework handles everything internally, ensuring uniformity in responses (in this case, Google Gson is the chosen serializer in the implementation). Even a simple String message is enclosed in quotes precisely because of the JSON serialization format.

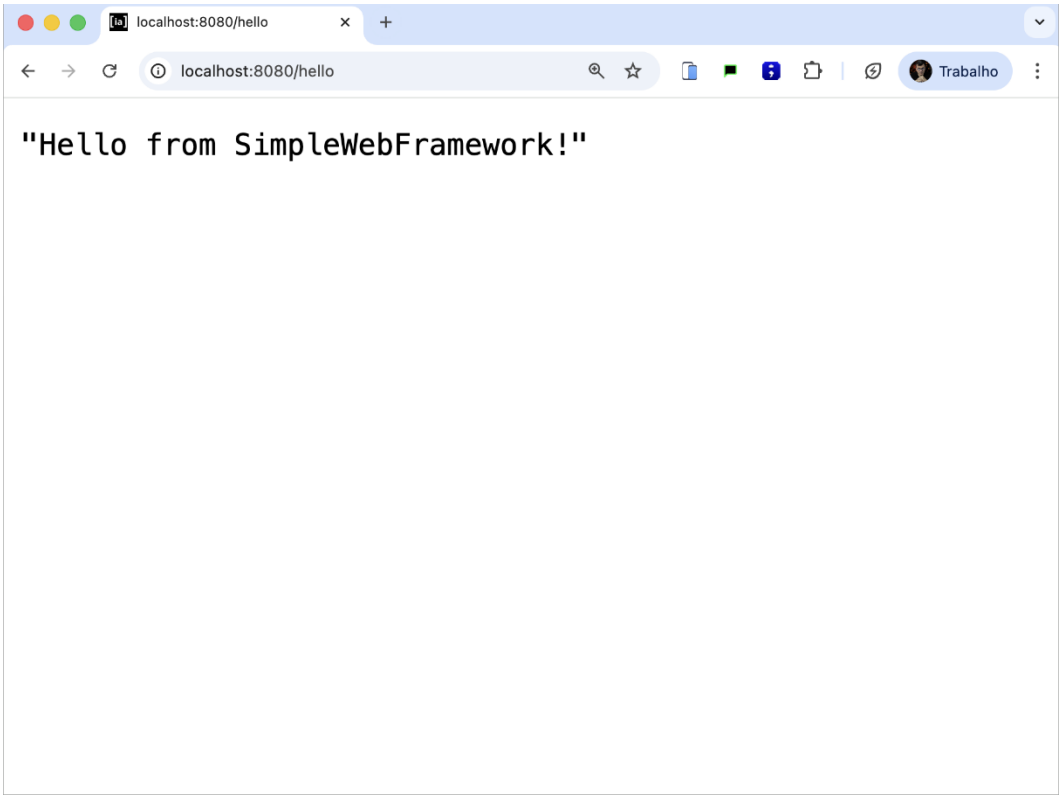


Figure 11.1: Simple "Hello world" output

The second method, `getProduct()`, returns an object of type `Product`, defined in the `Product.java` class. This class represents a simple entity, with `id`, `name`, and `price` attributes (and their respective getters and setters), and serves as an example of how the framework automatically handles more elaborate objects.

By accessing the `/product` route via the `GET` method, the framework instantiates the returned object and serializes it in JSON format, resulting in a response like the following:

```
{
    "id": 1,
    "name": "Keyboard",
    "price": 199.90
}
```

Here is a screen showing the browser performing this request via the URL:

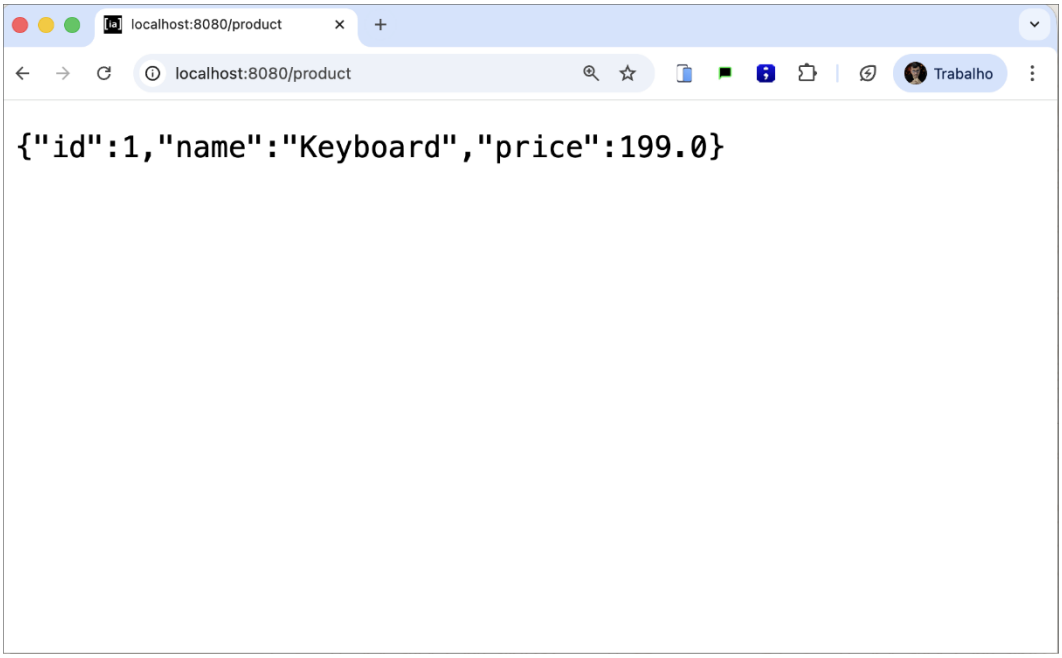


Figure 11.2: Browser output for GET request

This conversion is done transparently, without the developer needing to worry about formatting details, character encoding, or output streams.

The focus remains on the application logic, not the transport infrastructure.

Finally, the `createProduct(@SimpleBody Product product)` method demonstrates the inverse flow: sending data from the client to the server. Note that the parameter is annotated with `@SimpleBody` precisely to indicate to the framework that a string in JSON format must be read from the `HTTP` request body and converted to the parameter's type (in this case, `Product`).

When a `POST` request is sent to `/product` with a body such as:

```
{
    "name": "Wireless Mouse",
    "price": 129.90
}
```

The `SimpleWebFramework` interprets the body of the message, automatically creates an instance of the `Product` class, and populates its corresponding attributes. This Java object is then delivered to the `createProduct` method, completely ready for use.

Inside the method, the developer can execute any necessary logic – in this example, we simply assign an identifier to the product and return it.

The response sent to the client is again formatted in JSON.

```
{
    "id": 2,
    "name": "Wireless Mouse",
    "price": 129.9
}
```

As we can see in the following image made with the Postman application, the result of this POST request matches the expected result.

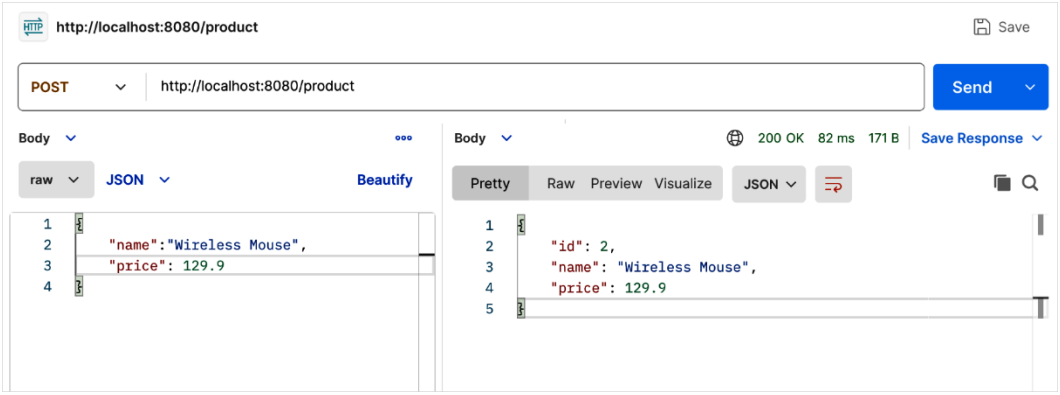


Figure 11.3: Postman interface for POST request

In this way, both sending and receiving objects follow the same uniform and intuitive pattern, without external dependencies or additional configurations. The framework handles all serialization and deserialization automatically, keeping the code clean, direct, and easy to understand.

Each of the three endpoints defined in this controller summarizes the central proposal of the `SimpleWebFramework`: to allow the developer to concentrate only on the application's behavior, letting the framework deal with all the technical aspects of HTTP communication and data formats.

Services and dependency injection in SimpleWebFramework

Throughout the previous chapters, we saw how the `SimpleWebFramework` has been built based on fundamental principles of modern software engineering, but always focusing on clarity and didactics. After understanding how controllers work, how HTTP routes are mapped, and how responses are automatically serialized, the moment has arrived to introduce one of the most important concepts in any contemporary web framework: dependency injection (DI) and the role of services within the application architecture.

In medium- or large-scale applications, it is common for the control layer – responsible for handling `HTTP` requests – to need to execute business rules, calculations, database queries, or integrations with external systems. However, putting all this logic inside a controller makes the code long, confusing, and difficult to maintain.

The good practice, adopted for decades in corporate frameworks, is to separate the business logic into service classes. These are specialized components responsible for executing specific tasks and providing results to the controllers.

This separation follows one of the most important principles of software architecture: Separation of Concerns. Thus, each layer has a well-defined role – controllers handle communication with the client, while services take care of business rules and the operations that support the application.

In the `SimpleWebFramework`, this same pattern is adopted in a simplified way, with the objective of transparently illustrating how professional frameworks, such as Spring Boot or Quarkus, apply the concept of dependency injection and component management (IoC containers).

For this, we will introduce two central classes: `IService.java` and `IServiceImpl.java`, which exemplify the creation and use of services within the framework.

The service interface: defining the contract

The IService.java interface represents the definition of a service contract. In practical terms, it describes what the service must do, but not how. This abstraction allows different implementations to be created in the future without altering the rest of the application, a fundamental idea of interface-oriented programming.

```
package com.myapp.services;

public interface IService {
```

```
    String getCustomMessage(String name);
}
```

The `getCustomMessage` method is simple but has a didactic purpose. It demonstrates that the service can receive a parameter and return a processed response – in this case, a personalized message.

By using interfaces, the developer creates a contract point between the control layer and the business layer. The controller does not need to know implementation details; it only knows that there is a service capable of generating custom messages. This decoupling reduces dependency between system modules and facilitates future extensions or substitutions.

The service implementation: linking the business logic

The `IServiceImpl.java` class is responsible for providing the concrete implementation of the IService interface.

It is marked with the @SimpleService annotation, indicating to the `SimpleWebFramework` that this class should be registered as a service component available for automatic injection. The implementation code is as follows:

```
package com.myapp.services;

import simple.annotations.SimpleService;

@SimpleService
public class IServiceImpl implements IService {

    @Override
    public String getCustomMessage(String name) {
        return "Welcome, " + name + "! This message was generated by
IServiceImpl.";
    }
}
```

The `@SimpleService` annotation plays a fundamental role. When the framework is started, it performs an internal process of searching for classes annotated with specific markers – such as `@SimpleController`, `@SimpleService`, `@SimpleInjected`, among others (as seen previously).

Upon finding a class marked with `@SimpleService`, the framework registers it in an internal data structure that associates interfaces with their concrete implementations. This registration is at the heart of the Dependency Injection (DI) mechanism.

This process creates a small injection container, that is, an internal repository of instances and metadata. It is thanks to this container that the `SimpleWebFramework` is capable of identifying which class should be instantiated and injected when a dependency is requested by a controller.

On-demand injection

One of the most interesting points of the `SimpleWebFramework` is that dependency injection does not happen in an anticipated or massive way, but on demand – that is, only when a controller needs to be instantiated to handle an HTTP request.

When the framework receives a request, it identifies which controller should be invoked. If this controller has not yet been created, the framework performs its instantiation. At this moment, it analyzes all the class attributes and checks whether any of them are annotated with `@SimpleInjected`.

When it finds this annotation, the framework searches in the internal structure for a class registered as an implementation of the corresponding interface. Then, it dynamically creates an instance of that class and assigns it to the annotated attribute.

The result is that the controller automatically receives all the dependencies it needs, without having to manually create instances or know implementation details. This is, in essence, the idea of Inversion of Control (IoC): the code does not create its dependencies, but receives them from an external mechanism.

The following example shows the practical use of this injection inside `HelloController.java`:

```
package com.myapp.controllers;

import simple.annotations.SimpleController;
import simple.annotations.SimpleGetMapping;
import simple.annotations.SimpleInjected;
import com.myapp.services.IService;
import com.myapp.model.Product;

@SimpleController
public class HelloController {

    @SimpleInjected
    private IService service;

    @SimpleGetMapping(path = "/hello")
    public String sayHello() {
        return service.getCustomMessage("Developer");
```

```
    }

    @SimpleGetMapping(path = "/product")
    public Product getProduct() {
        return new Product(1, "Keyboard", 199.90);
    }
}
```

In this example, the service attribute is declared as `IService` and marked with the `@SimpleInjected` annotation. This indicates to the framework that the responsibility for creating and associating the appropriate object belongs to its internal container.

When the `sayHello()` method is executed, the service has already been injected and is ready for use. The `getCustomMessage()` method of the `IServiceImpl` class is then invoked, generating a personalized message that will be automatically converted to JSON and sent as a response to the client.

Internally, the framework executes a process similar to the following pseudocode.

```
if (field.isAnnotationPresent(SimpleInjected.class)) {
    Class<?> type = field.getType(); // example: IService
    Class<?> impl = ServiceRegistry.getImplementationFor(type); // returns
IServiceImpl
    Object instance = impl.getDeclaredConstructor().newInstance();
    field.setAccessible(true);
    field.set(controllerInstance, instance);
}
```

This snippet illustrates in a simplified way the logic used by the framework: it identifies the type of the interface, searches for the registered implementation, creates an instance of that implementation, and injects the object into the corresponding field. This operation is done through reflection, a native Java feature that allows inspecting and manipulating class metadata at runtime.

As a result of all this, a simple endpoint now, in addition to triggering the corresponding route, also invokes the correct method to generate the expected message, as illustrated in the following figure.

Figure 11.4: Dependency Injection result

Understanding the didactic role of DI in SimpleWebFramework

It is important to understand that the dependency injection mechanism implemented in the `SimpleWebFramework` does not seek to reproduce all the complexity of corporate frameworks. Instead, it has an explicitly didactic purpose: to clearly show how the process works internally.

By exposing the behavior of injection, the framework helps the developer understand essential concepts that, in ready-made frameworks like Spring, are hidden under layers of abstraction.

In practice, this means that the `SimpleWebFramework` demonstrates the complete dependency injection cycle:

1. The discovery of annotated classes and interfaces
2. The registration of implementations in the internal container
3. The analysis of controllers at the time of instantiation
4. The dynamic creation of the necessary dependencies
5. And finally, the injection of objects at the correct point in the application

This simple but powerful process allows for a tangible understanding of what "Inversion of Control" means and how a framework can automatically manage the lifecycle of objects and their dependencies.

Field injection vs. constructor injection

There are different ways to implement dependency injection. The two most common are field injection (also called attribute injection) and constructor injection.

In `SimpleWebFramework`, field injection was chosen because it is more direct and visually clear for educational purposes. The framework simply locates the annotated attributes and injects the corresponding objects into them.

However, it is important to highlight that, in mature frameworks like Spring, the preferred approach is constructor injection. This strategy has significant advantages: it makes a class's dependencies explicit, facilitates unit testing (since dependencies can be simulated via mocks), and prevents the framework from having to directly alter private attributes, which could be interpreted as a partial violation of encapsulation.

In field injection, the framework needs to modify the value of private fields using reflection, which breaks the encapsulation barrier. Although it is functional, this approach is seen as less elegant in complex systems.

Nevertheless, within the didactic purpose of the `SimpleWebFramework`, this choice is intentional: it simplifies code reading and allows the student to clearly visualize the point where the injection occurs without the need for long or complex constructors.

Paths for SimpleWebFramework expansion

With the construction of the `SimpleWebFramework`, we reached an important milestone: the creation of a didactic and functional environment capable of handling the main elements that compose a modern web framework – route control, HTTP method mapping, dependency injection, and automated response in JSON format. However, like any modular software structure, the framework can (and should) evolve. Based on its solid and simple foundation, there are countless possibilities for expansion that allow it to broaden its capacity and bring it closer to professional frameworks, while still preserving the pedagogical character that defines it.

A first path of evolution is to improve the component lifecycle, allowing the developer greater control over how and when objects are instantiated. Today, the `SimpleWebFramework` performs on-demand dependency injection, meaning it creates objects only when they are needed. This behavior, although efficient for the current model, can be expanded with the introduction of lifecycle scopes, such as singleton and prototype, in the style of corporate frameworks. Thus, the framework could maintain a single instance of certain services throughout the application's

execution, saving resources and simulating the real behavior of more sophisticated IoC (Inversion of Control) containers.

Another promising aspect is the expansion of the serialization and deserialization layer, allowing the developer to choose different response formats, such as XML or YAML, in addition to JSON. This evolution would enable closer alignment with the concept of content negotiation, which is widely adopted in modern REST APIs.

But perhaps the most intriguing expansion is the creation of a new complementary module to the framework: SimplePersistenceAPI, a data persistence layer inspired by the functioning of the Java Persistence API (JPA).

This module's main objective would be to practically teach how JPA performs object-relational mapping.

The proposal would be to create a reduced set of annotations – such as `@SimpleEntity`, `@SimpleId`, `@SimpleColumn`, and `@SimpleTable` – to map Java classes to tables and attributes to columns. For example:

```
@SimpleEntity
@SimpleTable(name = "users")
public class User {

    @SimpleId
    private Long id;

    @SimpleColumn(name = "username")
    private String username;

    @SimpleColumn(name = "email")
    private String email;

    @SimpleColumn(name = "age")
    private Integer age;
}
```

The persistence layer could operate on CSV files or in-memory databases (like H2), abstracting SQL details and allowing basic operations, such as save, findById, and delete.

Internally, the SimplePersistenceAPI would show how Java can use reflection to automatically generate data storage and retrieval instructions, simulating the idea of a simplified EntityManager.

With this, it would be possible to offer a complete development experience: from the definition of entities and controllers to the persistent management of information, connecting all the concepts studied – HTTP, injection, serialization, and persistence – into a coherent and educational ecosystem.

The creation of a SimplePersistenceAPI, therefore, would not only represent a technical addition to the framework but also the natural step of a learning journey that seeks to reveal, in detail, the backstage of modern Java infrastructure, forming a developer capable of understanding the complete lifecycle of a web application, from the `request` to the database.

In this chapter, we could understand how to create a web application (or a microservice) using the Simple Web Framework. And your studies cannot stop here. You should keep evolving, working with web frameworks or web applications. But if you would like to contribute to this project, you can:

- Fork GitHub repository and make your contributions, such as changing the startup initialization (initialize all services and all controllers at startup time), improving the log class, and others
- Develop extensions for this framework, such as Simple Persistence API to connect to your preferred database
- Create a full production application using this Simple Framework to ensure that it is possible to deploy your application quickly and easily

I hope this material can help your personal and technical skills and make you curious to understand some of the internals of a technology that you use every day.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

12

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:

7-Day Packt Library Access
Get 7-day unlimited access to 8,000+ books and videos. No credit card required.
Available for first-time Packt+ trial users only.

Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **Note**
>
> **Note**: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 12.1* on desktop), search for this book by name and select the correct edition.

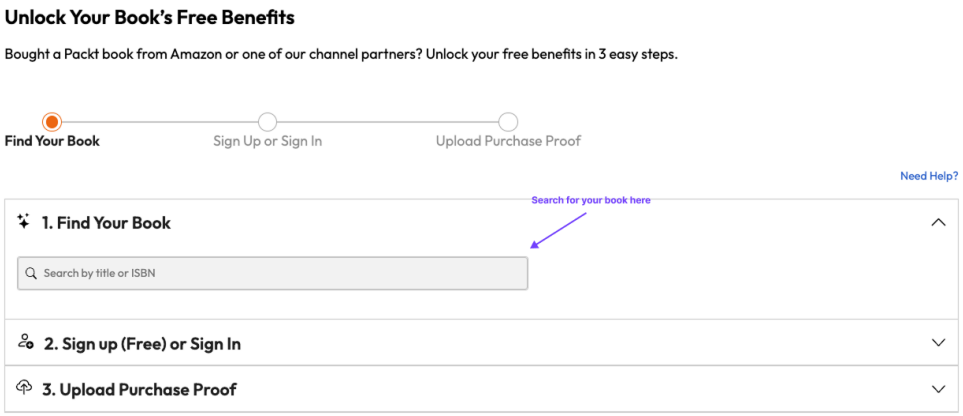

Unlock Your Book's Free Benefits
Bought a Packt book from Amazon or one of our channel partners? Unlock your free benefits in 3 easy steps.
Find Your Book
Sign Up or Sign In
Upload Purchase Proof
Need Help?
Search for your book here
1. Find Your Book
Search by title or ISBN
2. Sign up (Free) or Sign In
3. Upload Purchase Proof

Figure 12.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

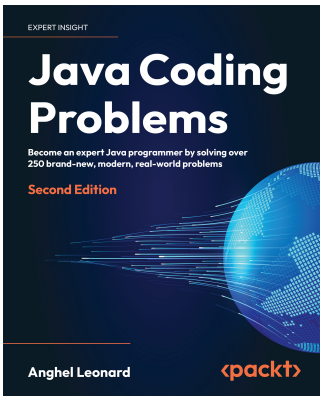
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

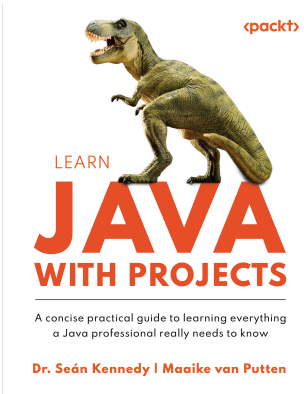

EXPERT INSIGHT
Java Coding Problems
Become an expert Java programmer by solving over 250 brand-new, modern, real-world problems
Second Edition
Anghel Leonard
<packt>

Java Coding Problems

Anghel Leonard

ISBN: 9781837633944

- Adopt the latest JDK 21 features in your applications
- Explore Records, Record Patterns, Record serialization and so on
- Work with Sealed Classes and Interfaces for increasing encapsulation
- Learn how to exploit Context-Specific Deserialization Filters
- Solve problems relating to collections and esoteric data structures
- Learn advanced techniques for extending the Java functional API
- Explore the brand-new Socket API and Simple Web Server
- Tackle modern Garbage Collectors and Dynamic CDS Archives


packt
LEARN
JAVA
WITH PROJECTS
A concise practical guide to learning everything a Java professional really needs to know
Dr. Seán Kennedy | Maaike van Putten

Learn Java with Projects

Dr. Seán Kennedy, Maaike van Putten

ISBN: 9781837637188

- Get to grips with Java fundamentals to build a strong programming foundation
- Gain a deep understanding of the critical object-oriented principles: encapsulation, inheritance and polymorphism
- Apply real-world scenarios using classes, objects, and interfaces
- Master exception handling for robust error management
- Explore generics and collections to manage complex data structures
- Utilize lambda expressions and streams for efficient data processing
- Complete practical projects to reinforce theoretical knowledge

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Java Web Internals*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

https://packt.link/r/1835881491

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.

https://packt.link/deep-engineering-newsletter

Index

Symbols

@SimpleInjected annotation **152**

@SimpleService annotation **150**

A

Apache Tomcat **95, 96**

application layer protocols
- need for 7

application server **55**

B

Bean Type **144**

C

ClassExplorer class **124**
- annotations, identifying with reflection 127
- automated discovery 128
- directory structure 127
- found controllers, registering 128
- role in initialization 125
- scanning, with DFS algorithm 125, 126
- search behavior 127

Contexts and Dependency Injection (CDI) **144**
- Bean Type 144
- implementations 146
- multiple expressions 148, 149
- structural decoupling 144 – 146

client-server communication model **7**

communication protocol **3**

connected sockets **10**

constructor injection
- versus field injection 173

D

DI in SimpleWebFramework
- @SimpleInjected annotation 152, 153
- @SimpleService annotation 150
- CDI 149
- ClassExplorer in service discovery 151
- decoupling and modular architecture 156, 157
- DependencyInjectionMap 153
- DispatcherServlet lifecycle integration 155
- implementing 149
- instantiation at startup 154
- instantiation on demand 154
- instantiation strategies 154

Dependency Injection (DI) **140, 144**

Depth-First Search (DFS) algorithm **125**

DispatcherServlet **99, 100, 113, 155**

Domain Name Service (DNS) **21**

dependencies
- challenge, in enterprise applications 139, 140

E

EJB container **142**

Enterprise JavaBeans (EJB) **140**
- Entity beans 141
- Message-Driven beans (MDBs) 141
- session beans 141
- XML configuration 142, 143

Entity beans **141, 142**

F

field injection
- versus constructor injection 173

framework **108**
- web frameworks 108, 109

G

GraalVM **97**

H

HTTP request **23**
- components 24, 25
- example 26, 27
- handling 40 – 42

headers 25

HTTP response 27
example 27 – 31
handling 42 – 50

HandlerAdapter 100

HandlerMethod 100

HttpMessageConverter 101

I

International Organization for Standardization (ISO) model 3

Internet Assigned Numbers Authority (IANA) 22

Inversion of Control (IoC) 140

J

JSP (JavaServer Pages) 95

M

Message-Driven beans (MDBs) 141

MyWebTestApplication project
creating 160

MyWebTestApplication.java file 161, 162

marshaling 11

metadata extraction module 113, 114
@SimpleBody 115, 116
@SimpleController 114
@SimpleGetRequest 115
@SimpleInject 116
@SimplePostRequest 115
@SimpleService 116

N

network communication 3

O

ObjectMapper 101

Open Systems Interconnection (OSI) model 3, 4
application layer 5
data link layer 4
network layer 4
physical layer 4
presentation layer 5
session layer 5
seven-layer model 6
transport layer 4

OutputStream 101

P

port 22

project setup, manual
client project, compiling 17 – 19
client project, running 17 – 19
JAR file, generating for protocol 17
server project, compiling 17
server project, running 17

projects, on Eclipse IDE
manual setup 16
setting up 15

protocol 21

Q

Quarkus 97, 98, 110, 147

R

RequestMappingHandlerMapping 99

resource
Internet Assigned Numbers Authority (IANA) 22

S

Session beans 141
Stateful Session Beans (SFSB) 141
Stateless Session Beans (SLSB) 141

Simple Application Server (SAS) 95, 99, 107
educational purposes, in educational environments 90
embedded server 90
expansions 84
improvements 84
logging 89
low-performance devices 92
monitoring 88
potential uses 89
review 84
session maintenance 85, 86
support for multiple applications or contexts 87
support for multithreading 84

tools, for administration and deployment 89
usage, in test environments 91

SimpleDispatcherServlet 129
ClassExplorer and DispatcherServlet relationship 129
consolidated flow 136
ControllersMap 130, 131
data structures 134
HTTP GET and POST methods, handling 135
internals 131
on-demand creation, of controller instances 134, 135
request body, reading with @SimpleBody 135
RequestControllerData structure 130, 131

SimpleWebContainer 65
architecture 57, 58
business rules, handling via HTTP 59 – 62
classes, exploring 70 – 74
example 75 – 78
example, testing 80
implementation issues 62, 63, 81

SimpleWebFramework 119
business logic, linking 169, 170
contract, defining 168
controllers 163 – 167
Dependency Injection (DI) implementation 149
on-demand injection 170, 171
paths for expansions 173 – 175
role of DI 172
route mapping 163 – 167

SimpleWebFramework architecture 112
browser to HTTP request 112
DispatcherServlet 113
embedded Tomcat 112

SimpleWebFrameworkApplication class 119 – 121
code 122
embedded tomcat dependency 121
embedded tomcat, benefits 124
framework initialization 121
framework initialization cycle 124

SimpleWebServer application
performance issues 50 – 52
starting 36 – 39

Spring 98
controller logic, invoking 101, 102
DispatcherServlet 100, 101
extensibility 102
handler mapping 99, 100
HandlerAdapter 100, 101
lifecycle 102
request interception 99, 100
response, generating 101, 102

Spring Boot 99

Spring Framework 109, 147

Spring MVC 99

server address 21

servlet 96, 97

simple application protocol
defining 8 – 10

sockets 10
connected sockets 10
example 11 – 15
unconnected sockets 10

structural decoupling 144

T

TCP sockets 10

TCP/IP model
application layer 6
media access layer 6
network layer 6
Transmission Control Protocol (TCP) 6
transport layer 6
User Datagram Protocol (UDP) 6

TCP/IP stack 2
exploring, with example 2, 3

Tomcat 56

U

Uniform Resource Locator (URL) 21

unconnected sockets 10

unmarshaling 11

W

WebContainerLib classes
creating 67 – 69

web framework development 103
DI, based on Jakarta specification 105
embedded Tomcat, using 103
GET method handling 103
object deserialization 104

object serialization 104
POST method handling 103
web frameworks **108, 109**
creating 110, 111
features 97
web server
internal structure 31
performance 32, 33